Trusted Execution for Private and Secure Computation: a Composable Approach

Lorenzo Martinico

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2024

Abstract

Trusted Execution Environments (TEEs) protect and isolate programs, sometimes referred to as enclaves, from all other software executed on the same processor, through a combination of specialised hardware, microarchitectural design, and cryptography. They are used both to underpin the security of computing infrastructure that processes sensitive data, and as a component in the design of efficient privacy-preserving protocols. The adoption of this technology is driven by a desire to shift trust from the operator of computing equipment to its manufacturer. A core feature of the architecture is remote attestation, allowing a remote party to verify that a TEE is running a certain program. While initial industrial deployments often used this to prevent sensitive data from leaking to untrusted clients, e.g. for media content protection, there is a growing interest in using TEEs to ensure that programs deployed on a cloud server will not run any unauthorised or outright mailcious computation on sensitive data provided by users.

TEEs are still a relatively novel technology, with security implications that are still not fully understood, and whose implementation has been shown to be vulnerable to many attacks. It is paramount to carefully analyse the guarantees offered by TEEs when used as a primitive in the construction of application protocols, and how multiple protocols relying on TEEs might interact with each other when composed. We approach this task using the tools of the cryptographic literature. In particular, we use the *universal composability (UC)* framework to examine a high-level formalisation of TEEs that supports composition without delving into the specific of implementation or different architectural choices.

To demonstrate the scope of the task, we follow the construction of a cryptographic protocol through several stages. We first extend and reformalise IRON (Fisch et al, CCS17), a protocol that provides secure computation in a cloud setting by implementing the *functional encryption* primitive. Our extension, called Steel, broadens the class of allowable functions a cloud server can compute, and we prove the security of our protocol in UC, by using the formalisation of TEEs provided by Pass et al (EUROCRYPT17).

We then question the cryptographic assumption of this formalism, showing that it doesn't adequately capture existing attacks which would make Steel insecure, such as state continuity attacks. We address this vulnerability by providing a modular trusted execution abstraction that can adequately capture TEE implementations with different functional and adversarial guarantees, and provides a more flexible attestation mechanism. Our new formalism aims to capture existing work in the literature and rescue previous proofs into a more realistic setting, closer to real-world implementations.

Finally, we show how the Steel protocol can be used as a composable building block to provide a privacy preserving contact tracing service that bridges the privacy properties of decentralised contact tracing with the ability to conduct data analytics enabled by centralised contact tracing. We construct and prove the protocol in a modular manner, showcasing the power of universal composability.

Lay Summary

As the saying goes, “the cloud is just someone else’s computer”. How do we know, then, that any data we are sending to the cloud is not being misused by whoever controls it? Trusted Execution Environments (TEEs) are a recent technology that could provide a promising solution. TEEs modify the design of traditional computer hardware to increase isolation between programs running on the same machines, and allow remote verification that the computer is running a specific program (a process called *attestation*).

Cryptography is a field of study that is broadly interested in rigorously proving the security of a computer system. It usually operates at a high level of abstraction, employing formal models and mathematical reductions, and relies on certain mathematical assumptions. TEEs provide an interesting tool for cryptographers, as they can prove that a computer equipped with a TEE “behaves correctly” through attestation.

In this thesis, we use state of the art cryptographic proof techniques to prove that TEEs can be used to compute any program without revealing the original user input, providing a *cryptographic protocol* (a precise recipe with a proof that it behaves correctly). We use our protocol as a starting point to develop a privacy-preserving contact tracing system for infectious deseases such as COVID-19, that allows health authorities to discover limited population-wide statistics if authorised by a sufficient number of users.

While our protocol is secure using the generally accepted abstraction for TEEs used by cryptographers, we argue that this abstraction does not adequately capture real-world TEE implementations, and as such does not capture realistic attacks on the system. We argue that more realistic models are needed to capture both what a TEE can do and how it is vulnerable to tampering. This would allow protocol designers to specify more realistic requirements, and we discuss how we can use this model to strengthen existing TEE implementations.

Acknowledgements

It is difficult to even begin thinking about everyone that has led to this work seeing the light of day. While research has certainly not been the only thing that I have done over the last 5 years, I do believe that every conversation, as unrelated as it might be, plays a part in subconsciously developing the required creative energy. If I could I would therefore like to thank many more people than what is reasonable for their warmth, affection and continued support. A special shout out goes to Adèla, Allie, and all the other flatmates across the last few years that have helped me make a home, however short-lived; to the Az/MOTW crew, for providing a (mostly) regular pressure valve for some mind wondering, nice chat and takeaways; the Musketeers, for being there when most needed and helping me get through the deepest slumps; to Aayush and Karis, for the challenging and fortright conversations, and the loving skepticism; to everyone at ESHC for the many valuable lessons and unique memories you have left me with; and to everyone who I had to explain to that (as much as I wished) being a cryptographer and a cartographer are not the same thing.

I want to thank everyone that has provided me with coaching and mentorship during my PhD, in particular Rix for doing her best to share her experience despite the very different settings, and Hazel for helping me make through in the last few months.

I am grateful to Muhammad Usama Sardar for showing interest in my work and encouraging me to share it, and to Pooya Farshim for the last minute feedback and insightful questions.

Within the school of Informatics, I would like to thank Patrick Hudson and Jonathan MacBride in the Graduate School for going above and beyond in helping me deal with University bureaucracy and their efforts in making all PhD students welcome at the school. I am very grateful to everyone in the Cryptosec lab and ICSA who have made working here a pleasant and convivial experience, and in particular I would like to thank the following people for letting me bug them for technical (and not so much) advice (in no particular order): Dimitris Karakostas, Orfeas Stefanos Thyfronitis Litos, Thomas Kerber, Aggelos Kiayias, Tariq Elhai, Christian Badertscher, Sabine Oechsner, Yu Shen, Paul Patras, Nadin Kokciyan, Dimitra Giantsidi, Jörg Thalheim, Maurice Bailleu, Jackson Woodruff, and Chris Vasiladiotis; Misha Volkhov and Christina Ovezik for keeping the crypto coffees alive (and really so much more!); Elizabeth Crites for making sure I survived my first conference; Michele Ciampi, for unexpectedly helping me discover some very relevant literature; Jan Bobolz, for some very last minute valuable feedback and warm words; Adam Jenkins and Mohammad Tahaei for always offering some valuable perspective; and Gareth Beedham, for helping make things happen.

I would also like to thank everyone who took part of my yearly reviews for the valuable feedback and encouragement: Vasillis Zikas, Mary Cryan, Pramod Bhatotia, Kami Vaniea and Myrto Arapinis. In particular, I'd like to thank Kami and Myrto for the always timely and relevant advice, and Mary and Pramod for pushing me when I most needed it and believing in me.

In a much more concrete sense, this work would not exist without all of the effort of my wonderful co-authors, who have had to endure so many iterations of my half-formed ideas and inscrutable notation, and have dedicated so much of their time on this work. There's too much to say, but I thank Aydin Abadi for always opening up his door for me at UCL; Thomas Zacharias, for the late night submission sprints and being honest about difficult topics; and Yiannis Tselekounis, for giving so much of his time to answer all of my dumb questions.

Most important of all, a massive thank you to my supervisor, Markulf Kohlweiss. I am grateful about the absolutely unreasonable amount of time and energy you have put over the years to ensure I would succeed, and for giving me the right balance of freedom and structure. I know it might not have felt like we got it right the whole time, but we got there in the end.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Lorenzo Martinico*)

Table of Contents

Chapter 1

Introduction

Trust is in the eye of the beholder

Balsa, Nissenbaum, and Park [45]

It seems that the first half of the 21st century is a time of low trust. The world is experiencing severe political and economic challenges that risk upending the geopolitical status quo. As misinformation and scams are rife, and historically unassailable institutions fail to deliver for the majority, many find themselves questioning previously unshakable beliefs. In an era of general mistrust, computer manufacturers have promised that *Trusted Computing* (also known as Confidential Computing [1]), a set of hardware technologies that can measure what software is running on a machine, as a measure to raise confidence in computers operated by remote parties. The implication of the technology should be wider-ranging and address pressing societal problems. But, in the words of philosopher Diego Gambetta, “can we trust trust”? Or Stated otherwise: Trusted Computing might be trusted, but is it trustworthy?

The adoption of cryptography in a system is often motivated by the argument that it replaces trust in human weak links, who are prone and can be easily incentivised to misbehave, with a rigorously proven cryptosystem. Just like game theoretical solutions can minimise the risk of misbehaviour in economic systems, cryptography can be used as a tool to enforce behaviour in a communication protocol, or minimising the fallout from misbehaving parties [2]. Breaking well-designed cryptography would require unobtainable levels of computation, such as a breakthrough in quantum computing, or finding a solution to mathematical problems that are generally considered hard to solve [3]. The promise of Trusted Computing is slightly different, in that trust is underpinned by the manufacturer of the technology, both in making adequate architectural choices, and in maintaining operational security during the manufacturing and deployment phases. Trusted Execution Environments, a type of Trusted Computing technology, specifically claim to isolate programs executed on commodity computers from other “untrusted” software, and to provide a hardware-based “root of trust” to verify which program is being executed, a process known as attestation. But how do these claims of trustworthiness square up to our common understanding of trust? And is it possible, or socially desirable, to solely rely on technically mediated trust, be it cryptographic or hardware based?

[1] In this work we will use the two terms more or less interchangeably, but Trusted Computing can be seen as applying more strictly to a subset of earlier Confidential Computing technologies (described in Section 2.2.1)[125]

[2] hence the motto adopted by some in the Blockchain ecosystem: “in proof we trust” [296]

[3] or, as critiqued by Munroe [212], a $5 wrench

The social context of trust. Humans are a social species, a fact that is made evident by how we organise in societies. As individuals, we are able to specialise, developing a small number of skills that we sell to the highest bidder, or freely give away. In return, we rely on the sum total of all other individuals' skill-sets and labour to ensure that our needs are met [261]. Over the last few centuries, this has led to the complexity of the highly industrialised and globally distributed economy that characterises today's world, where it is impossible for any individual or single geographical community to be completely self-reliant. In this context of co-operation, trust is to some degree a necessary component, which facilitates complex interactions that might otherwise lead to sub-optimal results[118]. It is the glue, or rather the lubricant [67] that binds societies together [192]. Indeed, high-trust societies seem to generally be characterised by stronger economic and social ties [133]. Assigning trust to someone however is an inherently risk-taking activity that involves the possibility of betrayal, and requires exposing oneself to vulnerability towards the trustee [41]. When we deem someone as trustworthy (or untrusworthy), we are assigning a belief on how likely they are (or not) to respect our offer of vulnerability and not use it to damage, despite being free to do so [118]. Yet, over the course of history we have spent much energy into devising strategies to minimise such exposure.

As the scale of these economic interactions have moved from highly localised in-person interactions in the physical world to global and digitally mediated ones, the success of a transaction can no longer be assessed by the social context in which it takes place. Online, are not able to use our finely tuned ability to read each other's body language and speech, or see and touch the goods we are purchasing. Rather, trust has to be mediated through infrastructure and intermediaries we might not have a first order relationship with, and/or through coercion and economic incentives. Regular computer users can find it hard to develop mental models of online system's trust [73].

The nature of trust is tied to the sociological context of the interactions it applies to, and as such has often changed across time and cultures [293]. The original architecture of digital communication system that have gone on to form the basis of our current internet infrastructure were designed for the purpose of freely sharing information among research institutions [177]. Reliability and interoperability were the primary concern, overshadowing any considerations of security, namely confidentiality, integrity and availability. It wasn't until more people and networks became connected to the nascent internet infrastructure, and practical demonstration of its vulnerability to abuse took place [223] that the threat began to be taken more seriously[4]. As lending trust to an entity is naturally an ongoing process that can be eroded over time[264], participants in the network lost their natural inclination to trust other agents. As a consequence, stricter security measures were taken in the design of future connect systems, and the term "trusted" began to refer to the components of a system on which the security policy relies on, and that can therefore lead to violations of the policy. Thereafter, the security and cryptography community, while attempting to minimise reliance into any one entity, began to refer to those minimal components of a system whose behaviour was expected not to deviate from a specific protocol as Trusted Parties [45].

Trusting the messenger (but not the message, or the interlocutors) As the last two decades have seen a dramatic increase of people and services connected through and exposed to the internet, security has become a crucial concern for the economic and political participation of people in both local and global society. The academia-driven free culture of knowledge sharing was first complemented and supported, and then replaced, by targeted advertisement. Commercial activity taking place online created economic incentives for algorithmic discovery, the data-driven expansion of the audience for any product, service or idea, in order to maximise the exposure to such advertisement, and the conversion into purchases. What began as a simple digital equivalent of a magazine classified ads based on the contextual relevance to the contents it accompanied, has tuned into an industry with a market valuation higher than most nations' gross domestic products[273]. This modern online advertisement ecosystem has turned the mass collection of (often sensitive) offline and online activities of pretty much anyone on Earth into a finely tuned automated behavioural prediction mechanism that gives an opportunity to the highest bidder to target groups based on highly specific demographics and interests. The precision of these inference systems can be high enough that many of its subject regularly complain that they are being listened to by their devices [131], whose software and hardware is often controlled by the same companies. This state of constant surveillance is identified by Zuboff [314] as Surveillance Capitalism, a new stage of market capitalism under which the extraction and processing of data from its subjects is the prime economic engine, with those generating the data being commoditised as resources rather than agents. This has become more evident in recent years, with the explosion in popularity of Large Language Models,

[4]Security concerns had already been identified by the military users, and led them to establish separate infrastructure [216]

which further made companies realise the value of data they already held as an asset. Both the earlier incarnation of targeted advertisement and its more recent "Artificial Intelligence" embodiment are at the root of a variety of issues, such as the algorithmic amplification of misinformation [174], that have caused widespread loss of social trust in online information.

While we might not have a reliable mechanism to trust the "message", some mechanisms have been shown to be effective to establish trust in the infrastructure that drives the delivery of that content. The construction of cryptographic and decentralised protocol has been a core feature of the last two decades of electronic communication, and has done much to advance our trust assurance from the previous paradigm of trust through authority or contractual obligation (privacy-by-policy [108]), which has repeatedly been chipped away by malicious actors [238] and misaligned incentives [167]. Narayanan [213] dubs this flavour of cryptographic work as *crypto-for-security*.

The philosophical underpinning of this process of "cryptographisation of trust" can be traced back to the cypherpunk movement [180] which, rooted in anarcho-capitalist ideas, proposed the adoption of strong encryption technologies as a tool of liberation, in sharp contrast with the prevailing doctrine of the US in the 1990s of classifying encryption technology as a munition. The cypherpunk's goal was to establish an ecosystem of "trustless trust", where activities that would have hitherto been mediated by trusted authority would be replaced by publicly designed protocols. Their clashes with western governments over the spread of cryptography as free speech has been referred to as the "crypto war", and resulted in the democratisation of encryption technology and decentralised protocols such as blockchains and "Web3" protocols. One could argued that the cypherpunk goal of pervasive upending of power has failed [213]: cryptosystems have not become such a crucial component in organising society, and most people's experience of digital technology still involves interacting with a number of trusted parties. On the other hand, it is also true that the use of cryptography has reached an unprecedent level of adoption in commercial systems[5]. In the last decade in particular, most major commercial communication services have now adopted some form of End-to-End Encryption, which prevents the operator of the service from reading or modifying the messages in transit between sender and receiver. End-to-End encryption is focused on removing trust from the infrastructure of communication services. However, once a message has been transmitted, there are no guarantees on what the receiver might do with the plaintext outside the constraints of the communication channel (or how

[5]sometimes at the cost of greater centralisation [211, 171]

the sender might have shared it outside that channel). Beyond the secrecy of private messaging, there are many compelling reasons to attempt addressing trust at the end-point of communication. While most consumer software requires the user to agree to stringent Terms of Service agreements, and most legislatures regulate the (mis)use of computers, the scale of deployments of some program makes it impossible to enforce these. Software techniques can be deployed to prevent specific uses of an application, and consumer device manufacturers are increasingly trying to lock-down their devices. A dedicated adversary can however often circumvent software anti-tampering measures through reverse engineering, leaking key material, network sniffing, or virtualisation. Any software attack can also be easily packaged and distributed to any other computer, increasing the value of deploying it. While it is tempting to refer to this kind of policy break by using the terminology of attackers and defenders [6], it is important to note that circumventing this kind of mechanism can be for the user's own benefit, as a way to escape artificial limitations added by a vendor, or to add essential accessibility features.

Hardware-based technically mediated trust Nonetheless, computer manufacturers have sought to introduce hardware protection measures to secure the software distributed by vendors. The advantages of a hardware protection mechanism include the ability to produce tamperproof hardware mechanisms that could stop functioning when an attack is detected, the increased difficulty in circumventing the countermeasures (compared to software ones) due to (more) specialised knowledge needed, and the loss of economies of scale due to the requirement of conducting physical attacks for each device. Despite the promise of increased security, the first attempts to produce hardware security mechanisms relied on the design of custom mechanisms, which were expensive to develop and deploy and could suffer from their own vulnerabilities. At the turn of the millennium, several technology companies, led by Microsoft, formed the Trusted Computing Group, in an attempt to create a standardised hardware security mechanism that would allow vendors to remotely prove the authenticity of a program running on any computer. The protection mechanism was originally designed to prevent the piracy of digitally distributed media, but as the consortium collected more members additional usages were proposed, from access control of sensitive data to enforcing authorised programs are run on a computer, especially for interacting with a remote system. The concept of trusted computing received severe backlash [18], with

[6]which cryptography inherits from its military roots

the Free Software Foundation dubbing it as "Treacherous Computing", as it would prevent the owner of computers from being fully in control of their own machines. In his talk "The coming war on general computation", Doctorow [120] outlined the risks of how commercial interests and political fear-mongering might lead to the adoption of Trusted computing as a censorship measure to prevent "undesirable programs" from running. Furthermore, concerns were raised on how limiting the authorisation of what kind of software could be run to a small set of companies could prevent a healthy competitive market and cementing the incumbents' dominance [19].

While the Trusted Computing Group's efforts were eventually scaled back, the ideas developed in that setting were taken up by successive vendor-specific efforts. In particular, underpinning the trust of general purpose computers to specific hardware features remained a popular idea. It is in this setting that CPU architecture vendors such as Intel started introducing the technology known as Trusted Execution Environments (TEEs), exemplified by the release of the Software Guard eXtension product (Intel SGX). SGX is a process isolation technology that uses microarchitectural features and cryptography to guarantee the integrity and confidentiality of code and the data it operates on. More concretely, it allows a CPU to run "trusted" code as a *secure enclave*, whose memory is inaccessible to other processes on the same machines. Enclaves can rely on Trusted Computing technologies to prove, in a process called *remote attestation* that they are running a specific program to a remote verifier.

It is difficult to draw conclusions on whether Trusted Execution Environments have been successful. On one hand, we can not point to any widespread commercial deployment of the technology on consumer devices. The original selling point of SGX was that it would allow a vendor to distribute sensitive information to an heterogeneous collection of untrusted consumer devices and prove that they would only execute authorised computations on that data. Realistically, the only use case that has been remotely successful has been the encryption of digital media (a practice known as Digital Rights Management) to enforce copyright protection (such as encrypting UHD video distributed through Blu-Ray disks or Netflix streaming content [231]). The lack of more useful applications, and the steady stream of vulnerabilities discovered by researchers, are likely among the reasons Intel discontinued SGX on consumer devices. At the same time, TEEs are still sold as an effective way to protect sensitive operations in the setting of cloud computing, with an increasing number of cloud vendors adopting the technology. This use case has captured the imagination of protocol designers, with many proposals to use TEEs as a replacement for a trusted third party to implement otherwise difficult cryptographic operations. Very recently, it has been advertised as a key component of Apple's strategy for providing privacy-first AI applications [1].

Position statement: Cryptographers and TEEs In his sweeping essay *The Moral Character of Cryptographic Work*, Rogaway [237] exhorts researchers to work on *crypto-for-privacy* problems that benefit everyday people, rather than the institutional state and corporate surveillance apparatus. In his view, cryptography can be an instrument to shift the balance of power. It is in that spirit that we argue, with this works, that cryptographers have a dual responsibility when it comes to the development of TEEs (and security technologies in general). In the first place, we reject the original goals of the Trusted Computing Group, which would take control away from regular computer users, and instead explore how the technology can be used to defend them against the abuse of more powerful (computationally or otherwise) "trusted" entities and force them to remain accountable. As such, our work is focused on the more recent trend of building protocols where users interact with cloud computing entities, who they rely on for performing computation, but do not trust to handle their data and might act adversarially. We see the introduction of TEEs in the server setting as an opportunity to shift the trust boundary from the server, replacing it with the guarantees provided by the combination of TEEs and cryptography (although there are some arguments for why client-side TEEs can still be pursued for legitimate applications [291]).

The other important role for cryptographers is to demystify the claims of commercial security vendors, and provide clean abstractions to validate that proposed solutions and applications are safely using these technologies. In a landscape where most deployable solutions are driven by industry for commercial motivations, it is hard to go beyond the marketing copy [7] and fully understand the characteristic of specific products from an objective perspective, when even the vendor themselves can not agree on definitions of what they are providing [245]. Being able to separate the behaviour of cryptographic primitives from their implementations is one of the essential requirements to reason about the wider constructions they might be component elements of. One of the techniques favoured by cryptographers is *Simulation-based security*, which proves the security of a scheme by comparing it to a high-level ideal object that behaves correctly. The work of Pass, Shi, and Tramèr [230] provided the first composable simulation-based idealisation of TEEs, boiling down its essence to the attested execu-

[7]another example of misuse of the term beyond Trusted Computing in the cybersecurity industrial complex is "Zero Trust"[139]

tion of arbitrary programs. Since their ideal model ignores most implementation details and the attacks that depend on them, one could argue that a proof in this model would not really be useful, as we can never fully realise such a perfect implementation. We take the position that using this type of functionality can still be meaningful, both to motivate research into better TEEs, and because there are already concrete steps we can take, through cryptography, to increase the assurances provided by the existing versions of TEEs.

At this point, it is essential to remark on the limitations that a purely cryptographic approach to our dual goals of redistributing power and demystifying security claims will face. First, we note that providing a purely technical solution, like we do, does nothing besides proving its possibility. For people to fully benefit from cryptographic designs, they require implementation and adoption. When, as in our case, the threat model includes service providers, it is not clear what their incentives for willingly adopting the system would be, when it might cause higher running costs, be more difficult to deploy and debug, and if their business model relies on surveillance capitalist principles. It is beyond the scope of this work to suggest regulatory or economic incentives to encourage the deployment of privacy-preserving trusted execution technologies. However, Balsa, Nissenbaum, and Park [45] argues that the adoption of privacy technologies by a service provider should be seen by the (potential) customers of privacy technology as a commitment to the value of privacy. An often overlooked limitations of establishing trust through cryptography is that end users of a service still need to blindly trust the cryptographers and security experts to appropriately design and assess the system, and that a deployment, which depends on multiple layers of infrastructure that are not always fully auditable [286], is successful. Establishing trust even in fully transparent software infrastructure is a historically hard problem [274], and there are very concrete concerns in particular around the transparency and auditability of TEEs in real world deployments [114]. There is a risk that, due to the gap between theoretical design and real world deployments, we might mislead users into trusting something that is ultimately not trustworthy.

1.1 Contributions and Structure

This work explores how TEEs can be used to re-establish users' trust in remote computing facilities they don't have direct control over, such as public clouds, to perform computation on sensitive data in a privacy-preserving manner. In particular, we follow the development of a specific TEE-based protocol to efficiently provide privacy preserving outsourced computation, with the goal of illustrating the necessary steps required in proving its security, as well as developing generalisable techniques that can be applied beyond the scope of this specific protocol.

After giving, in Chapter 2, a more in-depth review on relevant literature, and a list of pre-existing cryptographic definitions used in the rest of the work, we begin, in Chapter 3, by formulating a TEE-enabled protocol called Steel, a variant of an existing protocol [127]. Next, in Chapter 4, we show a weakness in Steel in real world systems, and explore how to capture more realistic models of attested execution and their relationship. Finally, in Chapter 5, we show how we can use a variant of Steel to construct a multi-user privacy-preserving computation to address a specific problem with wide-scale social implications around issues of trust. We conclude with Chapter 6 to highlight limitations and future directions of our work.

1.1.1 Steel: Composable Hardware-based Stateful and Randomised Functional Encryption

Cloud computing offers economies of scale for computational resources with ease of management, elasticity, and fault tolerance, driving further centralization of diverse applications into a small number of cloud computing providers. While cloud computing is ubiquitously employed for building modern online service, it also poses security and privacy risks. Cloud storage and computation are outside the control of the data owner (and sometimes the data processor) and users currently have no real mechanism to verify whether the third-party operator, even with good intentions, can handle their data with confidentiality and integrity guarantees.

There is a rich history of cryptographic work to enable secure computation of sensitive programs remotely. One such techniques is the primitive of *Functional Encryption* (FE), introduced by [59]. FE is a generalisation of Attribute/Identify Based Encryption [257, 242], that enables authorized entities to compute over encrypted data, and learn the results in the clear. In particular, parties possessing the so-called functional key, sk_f, for the function f, can compute $f(x)$, where x is the plaintext, by applying the decryption algorithm on sk_f and an encryption of x. Access to the functional key is regulated by a trusted third party.

FE is a very powerful primitive but in practice highly non-trivial to construct. Matt and Maurer [203] show (building on [9]) that composable functional encryption (CFE) is impossible to achieve in the standard model, but achievable in the random oracle model. For another important variant of the primitive, namely, *randomized functional encryption*, existing constructions [8, 146, 169], are limited in the sense that they require a new functional key for each invocation of the function, i.e., decryptions with the same functional key always return the same output. Finally, existing notions of FE only capture *stateless* functionalities. Motivated by the inefficiency of existing instantiations of FE for arbitrary functions, Fisch et al. [127] propose Iron, a practically implementable protocol that realises FE using a TEE, and is proven secure in the game-based setting.

Taking Iron as a starting point, we propose to deal with the above limitations by constructing a TEE-based protocol that allows to compute FE for a *broader class of functionalities* under the strongest notion of *composable security*. Namely we define a generalisation of Functional Encryption to arbitrary *Stateful and Randomised functionalities* (FESR), that subsumes multi-client FE (which allows multiple parties to independently generate ciphertexts [96]) and enables cryptographic computations in a natural way, due to the availability of internal randomness. We provide a definition of FESR using Universal Composability (UC) [75], a simulation-based proof technique that allows reusing ideal functionalities as building blocks of bigger protocols, without having to prove the security of each implementation in its new context. We extend the protocol of Fisch et al. [127] to capture the new notion into the Steel protocol. To prove that Steel realises FESR, we uplift the TEE idealisation of Pass, Shi, and Tramer [229] into the latest version of the standard UC model [75, Version of 2020], using the Universal Composition with Global Subroutine (UCGS) extension of [37]. Our security proof shows that one can satisfy a FE ideal functionality as defined in [203], relying on hardware instead of Random Oracles, and for the larger function class of FESR when additionally relying on a common reference string.

1.1.2 AGATE: Augmented modelling of Global Attested Trusted Execution

The PST functionality provides a clean abstraction for TEEs that facilitates security proofs of protocols using Universal Composability. By necessity, such a high level formulation should not contain precise implementation details for any one TEE platform. Given the vast number of attacks on TEE implementations, it is necessary to question whether the promised guarantees can be actually delivered. This is an issue that most protocol designers that incorporate TEEs in their constructions conveniently choose to ignore, leaving platform vulnerabilities such as side-channel attacks as out of scope. While in principle this approach is justifiable, as it would be unreasonable to ask cryptographers to become experts in the finer details of computer architectures necessary to create TEEs, a more realistic model is warranted if we are to see the deployment of these protocols in the real world. Replacing the idealisation of a TEE with a specific instantiation is bound to invalidate any security claim. We give a salient example by showing how a weakened abstraction that allows malicious adversarial interference with an enclave's state could lead to loss of confidentiality in Steel, by mounting a *rollback attack*. Previous works [278, 122] have shown that, for some protocols, a (significantly) weaker TEE implementation can still provide meaningful guarantees. The existence of these works suggests the need for cryptographers to articulate more precisely what aspects of a TEE their protocols will rely on. Articulation requires an appropriate language; our goal for this chapter is to create one.

We augment the ideal PST functionality with three "units of meaning" that can be modularly selected to provide different TEE guarantees: features, attacks, and attestation contents.

Features model the high level (trusted) interface available to programs executed within a TEE to interact with the untrusted portions of the machine or the outside world. The implementation of a feature might be implemented through a specific hardware modifications to the CPU architecture, trusted firmware, a cryptographic protocol between multiple enclaves and remote parties, or a combination thereof. As such, we give the enclave program access to "oracles" (an abstraction of a trusted interface) for the available features.

Attacks are also represented as abstract oracles, available to the adversary when interacting with the ideal TEE functionality. When constructing protocols that interact with TEEs, the attacker is generally modelled as the party that is executing an enclave on their local machine. As such, we give the attacker the option of passing additionally malicious control instruction along with any input to the enclave, and explicitly state in the formulation how a call to that oracle will affect the internal enclave state.

The content of Attestation that are transmitted to a remote verifier to certify the authenticity of the installed program are defined as a function over the state of the enclave (its *measurement*) and is bound to the TEE instance it runs on. The PST model has a rigid definition of attestation, with its guarantees inspired by the earliest scheme adopted by Intel SGX. Our formulation is more abstract and allows us to adopt a wider class of measurements and attestation properties.

Our modelling of these interfaces is presented in a modular fashion, with a shared baseline abstraction that provides an interface to parties interacting with TEEs. For each instantiation of a TEEs, we capture its unique combination of features, attacks and attestation through a combination of UC "shells", a modelling construct that allows us to reason about the interface of the enclave without the need to analyse the specific applications it is running. We provide several examples of shells that capture pre-existing formulations of TEEs in the literature, unifying all previous PST variants.

By providing a modular functionality for TEEs, we let the security proof for a protocol be independent from a concrete TEE instantiation. The protocol designer simply needs to provide a lower bound on what features the enclave programs require, an upper bound on how an attacker is allowed to tamper with enclaves, and how much information about the enclave is provided to other parties (or "leaked" to the environment) by the attestation. Despite this, we do not want to dismiss the pre-existing work to prove protocols as secure in the simpler PST model (including our proof of security for Steel). As such we propose a technique to bridge different versions of the functionality, either by adding a new feature oracle, or by removing an attack oracle. We show how to construct generic "wrapper" protocols which, combined with a less powerful TEE abstraction, are functionally equivalent to a stronger one, by implementing the missing features in runtime, or patching the remaining attacks. Showing that a more realistic TEE formulation, combined with the appropriate wrapper, is equivalent to PST could allow us to preserve pre-existing proofs under Universal Composability. By repeatedly showing that the combination of a "weak" TEE with a protocol implements a "stronger" TEE, we can provide a path to realise a powerful abstraction such as PST from realistic TEE implementations. We hope that our functionality will provide the cryptographic community a unifying abstraction to characterise different versions of TEEs, including those that have already been proposed in the literature, and will help ananlyse how they relate to each other. This is an important step to enable a more nuanced discussion on the security claims of TEE vendors and the requirements for TEE-enabled protocols - but is ultimately still a concern for theorists.

1.1.3 Glass-Vault: A Generic Accountable Privacy-preserving Exposure Notification Analytics Platform

Our last chapter attempts to satisfy our other call to action of providing protocols that shift the existing balance of power. Our choice of application is regretfully contingent to the historical circumstances of when this work took place, and we hope that our work will not need to be put into practice soon. As countries across the world were ravaged by the COVID-19 pandemic in 2020, a natural response for academics, who like many at the time had their personal and professional lives deeply affected, was to reflect on what role they could play in addressing the loss of life and rapid changes in the way society operated. Cryptographers in particular became involved with the public debate on what would be appropriate ways to use digital technologies, such as smartphones, to automate the process of contact tracing (identifying individuals who were in close proximity to infectious people) and exposure notification (notifying them of such contact). This Chapter represents our contribution to an already crowded field, in an attempt to find a balance between giving people control about what data they are willing to share, and providing public utility.

For a disease with a high risk of transmission among people in close proximity, contact tracing and exposure notification can be crucial mechanisms to diminish its spread[91]. Identifying and instructing only those who have potentially contracted the virus to self-isolate, removes the need to require an entire community to lock down. This is crucial when combating a pandemic, and early adoption of an adequately designed system could have prevented the destructive effects on people's (mental and physical) well-being and countries' economies that followed the numerous lockdowns imposed by health authorities. As manual contact tracing requires a ramp-up for hiring and training case workers, and is error prone when contact can be made from a distance and short periods, researchers have proposed smartphone-based solutions to improve data collection through automation. Most of the proposed solutions, in part due to the participation of cryptographers, attempt to implement privacy-preserving techniques, such as hiding the contact graphs of infected users or adopting designs that prevent tracking of non-infected users [198]. The role of privacy in those designs was both a value in and of itself, but was also considered a necessity to engender sufficient adoption throughout the population [152]. Besides Bluetooth generated contact information, there have been a few ad-hoc privacy-preserving solutions that help *analyse* other user-originated data, such as location histories to map virus clusters [66], or QR code scans to notify anyone who might have been in the same location as infected individuals [191].

This kind of data analytics, especially if combined with public health records and wider population statics, can play a crucial role in enabling governments to take effective and proportional decision making and enhance public health advice. Despite the importance of this type of solutions, only a few proposals can also preserve user privacy, and even those suffer from two important limitations.

Firstly, they *lack generality*, in the sense that for each analytic task, different data encodings have to be sent to the analyst, which ultimately (i) limits the applications of such solutions, (ii) increases user-side computation, communication, and storage costs, and (iii) demands a fresh cryptographic protocol to be designed, defined, and proven secure and private for every operation type. The solutions proposed in [66, 191] are examples of such ad-hoc analytic protocols. Instead, it is desirable that different kinds of user data can be securely collected and transmitted through a unified protocol. In concrete terms, such a unified protocol would (a) provide a generic framework for scientists and health authorities to focus on the analysis of data without having to design ad-hoc security protocols and (b) relieve users from installing multiple applications on their devices to concurrently run data capturing programs, which could cause issues, especially for users with resource-constrained devices.

Secondly, existing solutions *lack accountability*, meaning that users are not in control of what kind of sensitive data is being collected about them by their contact tracing applications. In some of these schemes, the data does not even originate directly from the users, but from a third-party data collector, e.g., a mobile service provider, a national health service, or even national security services [16]. Even when health authorities attempt to limit the access to raw health data for research purposes [14]. individuals are not generally given the possibility to explicitly choose and withdraw consent on how much and how their data is being used. As privacy legislatures across the world have increasingly begun to recognise [93] this right, we need to develop solutions that give patients this level of accountability, while also providing sufficient analysis for public health purposes.

To address the aforementioned limitations, we propose GlassVault, a new cryptographic protocol. GlassVault is an extension of regular privacy-preserving decentralised contact tracing, where decentralisation roughly means that no central records are kept about uninfected users. GlassVault additionally allows users to share sensitive (non-contact tracing) data for analysis.

In this context, a functional encryption scheme such as Steel offers a compelling mechanism to address the problem of generality, providing a generic interface to authorise the computation of arbitrary functions on encrypted data, with the authorisation phase crucially not being required before the data collection phase. The question of accountability remains unresolved though, as decryption authorisations are entirely controlled by a trusted authority. The role of the authority is to establish the trustworthiness of entities (which in this setting we denote as analysts) requesting function evaluations, and the kind and scope of functions that should be authorised for a given level of trust. An obvious option for the role would be that of a data protection authority, which can investigate the data protection practices of organisations and levy fines in case these are violated. This solution would not actually give any direct power to the data subjects, and is ultimately another instance of privacy-by-policy. We propose, following previous work [96, 4], to democratise the decision of what analysts are authorised to compute. We remove the FESR key generation role from the trusted authority, and replace it with a threshold mechanism among the encrypting parties (anyone who is sharing personal data with the system). Finally, it is necessary to replace the fixed setup phase with a mechanism for parties to dynamically join as participants (as new users are enrolled). We provide a formal definition for this new primitive of decentralised and dynamic functional encryption DD-FESR, and extend the Steel protocol into the corresponding extension DoubleSteel. As the protocol and functionality maintain similarities with the simpler setting of Chapter 3, our proof of security is economically constructed in a modular fashion, reusing components of the previous proofs where the two protocols are the same.

The construction of GlassVault is thus a simple combination of an exposure notification, of which Canetti et al. [84] provides a UC formalisation, with an instance of DD-FESR, where all of the users of exposure notification correspond to the DD-FESR encryptors, and analysts (e.g. government health authorities, universities, or non-profits) take the role of decryptor. The proof of security for GlassVault justifies our choice of Universal Composability, by letting us reason at the level of ideal functionalities, without requiring to delve further into the implementation details of the relevant protocols.

As GlassVault provides a powerful general platform to compute data in the context of a contact tracing protocol, we believe it provides an important bridge between the paradigms of centralised and decentralised contact tracing, addressing the concerns of both public utility and privacy.

1.2 Notation

With any type of work trying to describe a phenomenon or idea formally, there is a constant tension between precision, readability, and succinctness. Prioritising one of these dimension is often a matter of taste and accepted convention. Among many works in the cryptographic literature, particularly around the vein of Universal Composition, precision often takes a back seat to succinctness, and readability (with varying degrees of success). This can, however, cause uncertainty on the authors' intentions and requires both some amount of extrapolation on the semantics of certain operation, and filling gaps in the description when certain steps are omitted as obvious (to the authors). While this could be seen as a foil preventing misuse that might stem from a surface reading of the work, it is an accepted [20] issue in the security community that many real world security issues arise between the gap of an academic cryptographic design and its implementation by practitioners.

In this work and the constituent publications that form its later chapters, we have opted to prioritise formal precision to minimise the number of ambiguities. While this might sometimes come into conflict with readability and certainly succinctness, we provide corresponding explanation in prose whenever a more precise algorithmic definition is presented. We now explain some of the conventions used in our pseudocode in the remainder of this work.

Data Types We represent boolean values with $\top, \bot$ for True and False (sometimes $1, 0$). We use standard notation for generic logical operations. Other standard datatypes include Strings, natural numbers and bitrsings (sequences of 0s and 1s).

Variables are empty until a value is first assigned to them, using notation $x \leftarrow v$. Comparing an empty variable with $\bot$ (or ε for strings) returns True. When accessing data from a variable, or calling a function, if we are not interested in the return value (or a partion thereof), we assign it to $\cdot$.

Data Structures A tuple (or pair) is an ordered collection of data of fixed length (at least 2). Its elements can be of distinct data types. We construct a tuple τ containing elements α and β (in that order) using the syntax $\tau \leftarrow (\alpha, \beta)$

A list (or array) is a mutable length ordered collection of data of the same data type. We construct an empty array a with syntax $a \leftarrow []$, and new element α is added via $a \leftarrow a||\alpha$. Our syntax for accessing elements of the array is inspired by programming languages like Python, which allows array indexing and slicing through the same syntax. In particular, arrays of length $|n|$ can be accessed via an integer $\{0,\ldots,n-1\}$ from "left" to "right" (i.e. in the order its elements were added), and via integers in $(-1,\ldots,-n-1)$. An array slice operation creates a smaller array that includes a subset of the original one. We use syntax $a[i:j]$ to denote the array a slice containing all elements from index i to index j (exclusive). We can also create a slice by using a logical operation: for example $a[i>0]$ will return an array that contains all elements of a such that $a[i]>0$. To generate a list of all integers from 0 to n, inclusive, we use the notation $\{0,\ldots,n\}$.

A set is an unordered data structure that contains a single copy of all its element. Inserting an item in the set if it is already included does not modify it. An empty set is denoted by $\emptyset$ or $\{\}$.

A dictionary (or map) is a data structure that links values to a unique key. We use the notation $v \leftarrow d[k]$ to indicate that dictionary d stores value v under key k. We can also access value v using the $d.k$ notation. An empty dictionary is denoted by $\{\}$. A dictionary can store any data structure as its values. Keys are generally integers or strings, but can also be tuples. In that case, we generally use a comma separated list of values as a key e.g. $d[l,r]$. We allow partial matching on a dictionary's index by using the notation $c \in a[b,\cdot]$, which returns value c stored for a key tuple whose first item is b.

We refer to Sections 2.1.2.3 2.3 for additional notation, and the introduction to Chapter 4 contains some remarks about the notation used in that Chapter. Additionally, whenever a proof defines some additional notational shorthand, it is defined before the pseudocode listing.

Chapter 2

Background

I don’t like SGX at all

Thomas Ptacek

This chapter provides an overview of the fundamental prerequisite concepts on which the rest of the works rely on. We begin with Section 2.1 by giving an overview of the tools used by cryptographers to rigorously show security, with a particular focus on Universal Composability, and related extensions that we rely on. Next, in Section 2.2, we describe the broad topic of Trusted Execution Environments. We begin with some definition of terms used in this field and historical definitions in Section 2.2.1, and give a high level description of Intel SGX, a popular TEE, in Section 2.2.2. In Section 2.2.3 we give pointers to the broader literature on TEE applications and attacks, with a particular focus on surveying the scope of State Continuity attacks (Section 2.2.3.1). In Section 2.2.4 we summarise previous efforts in formal modelling TEEs, including a detailed overview of the PST Universally Composable formalisation. We conclude the Chapter by giving, in Section 2.3, an overview of other cryptographic primitives and functionalities used in the remainder of this work.

2.1 Provable security

Provable security is an area of study, with its roots in the cryptography community, that is broadly concerned with systematically showing that a cryptographic algorithm or a system of interacting programs operates in its prescribed manner even in the presence of malicious entities. While a variety of other disciplines share its goals, such as computer security, formal methods, and distributed computing, what distinguishes provable security is the rigorous definition and specification of both the cryptographic object under scrutiny and adversarial power. A cryptosystem can be said to be provably secure if it can be shown to be equivalent to some security definition through a mathematical reduction to some fundamental computational hardness properties. As the scope of provable security broadens, it also has to incorporate concerns from some of the other aforementioned disciplines and find ways to fit their concerns into cryptographic abstractions.

The methodology selected to assess a system greatly influence both the scope and strength of the established guarantees, as well as the ergonomics of producing the evidence itself. For the purposes of this work we do not broadly consider techniques popular in software engineering like automated testing, or the problem of assessing whether a program correctly implements its specification, because we are interested in providing higher level notions on how to design secure and trustworthy systems, rather than assessing a specific implementation. However, we will remark when some of the techniques discussed below can be applied to those setting or when discussing relevant previous work. Rather, we consider exclusively work that primarily models message passing between different entities as abstract processes, rather than code running on real or simulated computer hardware and network.

2.1.1 Designing Provably Secure Protocols

We define a cryptographic protocol as a series of algorithmically prescribed operations between a set of distinct computational processes, the principals (or parties). A protocol is said to be secure if it satisfies certain guarantees, regardless of a subset of the principals diverging (in a well defined manner) from their prescribed role, or any attempts from a third party to tamper with the communication between the existing principals. The misbehaving principal and tampering third parties are known as adversaries.

A proof of security for a cryptographic protocol requires a model to represent computation and adversarial powers. The two models most used by cryptographers are the symbolic, and computational model. The symbolic model (also known as Dolev-Yao, formal [2], or term-based model) represents computation as a series of objects, with a well-defined interface that a principal or adversary can interact with. A cryptographic primitive is an object that can be instantiated by an adversary, but does not allow them to obtain its internal state or any other information beyond what is provided by the usage of its interface (a black box). Each primitive might be a term in some associated relational equations that show what partial information can be gained from interacting with it using different inputs. This level of generality easily allows automating the verification of cryptographic properties of a protocol in a non-deterministic setting.

The Computational model represents protocols as executions of Turing Machines, an idealised version of a computer which manipulates bitstrings. In this setting, a cryptographic primitive is a function between bitstrings. All principals, including the adversary, are Turing Machines. The Security parameter represents the length of the adversary's input string, which determines its total runtime and memory, and the size of keys using in the protocol. While security in the symbolic model is a black and white statement on whether by choosing a particular sequence of invocation of primitives the adversary can break any properties of the protocol under examination, a proof in the computational setting is an argument on the probability of an adversary (not) fulfilling some task. In most cases a security proof will argue that a successful adversary would have to violate some *computational hardness assumption* i.e. compute a program for which we do not have efficient solutions, or guess a value from a sufficiently large domain. There are open questions on what type of hardness assumptions are reasonable to adopt [142], and what kind of cryptography would be impossible if we realised that certain assumptions were not true in our world [156]. Certain protocols can be proved to be secure without relying on any hardness assumptions; we call this property *information-theoretical (or unconditional) security.* While the Turing Machine might be a simplistic model of computation, and the computational model ignores most low level hardware or physical modelling that could be exploited by an adversary in a real life physical machine, it is still a closer representation on how computing devices execute operations than the black box objects of the symbolic model, and as such as any attacks in the symbolic model can be translated into the computational model (the converse being much more difficult) [2].

A tool in either symbolic or computational models can verify security properties based on an execution trace, or argue on the equivalence of the protocol with a specification [57]. For proving equivalence properties in the symbolic model, several types of process equivalences exists [112]. In the computational model, security arguments can be structured as a decision problem for the adversary to break some property of the system, or as a proof of equivalence between the system and its specification In the first setting, the proof is formulated the game between the adversary and another entity (sometimes known as the challenger). The adversary wins the game if it can produce a certain outcome in its interaction with the challenger with some probability. Security is usually shown through a sequence of games, starting from the definition, to a game where the property holds trivially. Each game transition (or game-hop) should be atomic, with the probability of the adversary succeeding being the same or within a small boundary for each step of the game[258].

In the computational model we often refer to equivalence proofs as indistinguishability from an *ideal functionality*. This is an unrealistic perfectly trusted third party that takes the input of each party protocol and honestly computes the outputs, providing a perfect specification for the protocol. An equivalence proof for a protocol formulates a simulator, a machine that acts as the adversary in this ideal world and replicates the traffic of the real-world protocol, including the attacker's behaviour. Then, through a series of reduction assumptions, the cryptographer can show that real and ideal world are computationally indistinguishable i.e. any attacks that the adversary can carry in the real world must also be allowable by the ideal functionality. A common reduction technique is to provide a sequence of intermediate simulators from the ideal to the ideal world (or vice-versa), where each step can be justified to be indistinguishable on its own using game-hopping [189].

We refer the reader to Barbosa et al. [48] for a survey of techniques for automating security proof in the settings described. Next, we focus on a specific proof systems for simulation-based security we will be using for the remainder of this work.

2.1.2 Universal Composability

A significant challenge when modelling the security of a certain protocol π is the ability of capturing how one execution of π interacts with other protocols. If a single copy of π can be run securely multiple times, with no interactions with other machines, we say that it is secure under self-composition; when multiple protocols co-exist and interact with each other, security is proven under general composition. Other factors to take into account are the number of executions that π supports, whether the set of parties running the protocols are fixed or unbounded, and how each instance of π is scheduled (sequentially, when each run begins after the previous one has terminated; parallel, when multiple runs begin at the same time; or concurrent, where multiple runs can be arbitrarily interleaved [188].

A variety of theoretical frameworks to guarantee composition in one of these dimensions have been formulated [17, 31, 207, 206, 204, 165, 141, 74, 235, 113], but the most popular one to jointly guarantee all three types of composition in the computational and indistinguishability setting is the simulation-based Universal Composability framework (UC) proposed by Canetti [76] (developed concurrently to the work of Pfitzmann and Waidner [233]).

UC is based on the computation model of Interactive Turing Machines (ITM) [143] and allows constructing simulation-based proofs of security in a modular way. Due to its flexible modelling of communication channels and adversarial capabilities, UC can capture a broad variety of adversarial scenarios, and a large number of protocols have been shown to be UC-secure. Moreover, since its introduction the framework has inspired numerous extensions and variations [205, 32, 173, 153, 72, 78], including different revisions to the original model (see [75, Appendix B]).

2.1.2.1 Fundamentals

A succinct but comprehensive summary of the key components of UC can be found in [37, Section 2] and in [54, Appendix B]. In this section, we give a higher level overview of the model and discuss conventions adopted in the rest of this work.

A protocol is defined in UC as a set of ITM instances (ITIs) whose unique identity is composed of a party identifier (PID) and a shared session identifier (SID). We generally refer to the ITIs that represent the protocol principals as main parties, which can spawn subroutine that represents portion of code executed by the principal. To allow separating modelling artefacts from the code of the analysed protocol, a "structured protocol" divides ITIs into a shell and body component (introduced in [75, Version of 2018]). The body of the protocol handles the cryptographic operations, and is not aware of the shell, which is limited to handling modelling related instructions and can read and modify the contents of the body appropriately. A protocol is executed in the presence of a probabilistic polynomial time (PPT) bound machine, the environment, that captures the influence of any computation that might be taking place outwith the current instance of the analysed protocol. The environment can be seen as initialising the computation of the protocol, and providing input to each of the protocol principals and the adversary. The adversary is another PPT-bound machine that is able to instruct ITIs with special corruption messages to modify their behaviour, through a dedicated *backdoor tape*. For the rest of the paper, we assume the convention that any adversary is a *dummy adversary*, where its behaviour is to simply forward corruption messages originated by the environment to protocol parties. Besides the adversarial backdoor tape, ITIs are able to communicate with each other by writing messages on some dedicated tapes. These mechanisms should not be seen as equivalent to network communication but rather as a modelling artefact, while the network model can be implemented as an ideal functionality (allowing flexibility to model networks with different properties). While the framework does not impose general restrictions on which ITIs can communicate with each other, there are certain communication topologies that can be considered "better-formed", and necessary for certain composition results (such as subroutine respecting protocols, where all communications to protocol subroutines have to originate from the protocol main parties or one of their subroutines). To allow composing our examined protocol, the environment represents external communication by claiming an external machine's identity when sending an input to the protocol parties. An environment is said to be ξ-identity-bounded if the set of identities it can claim is restricted by ξ (expressed as a predicate over the system's state at the time the environment sends a message claiming an external machine's identity).

The model of execution of ITI is inherently single-threaded, but allows flexibility in describing the granularity of operations and how they interleave. Runtime constraints are satisfied by maintaining a runtime budget for each machine (known as *import*). Import can be shared with a machine's subroutine, allowing arbitrary dynamical subroutine nesting without running the risk of exceeding the remaining runtime. The minimum import considered by UC protocols is the length of the security parameter. A *balanced* environment ensures that at any point during the execution of a protocol, the adversarial import is at least as large as the sum of imports for all other ITIs in the protocol.

Like other simulation proofs, the basic mechanism for showing UC-security is to define an ideal functionality, which captures the essential properties of the desired protocol as being run by a trusted party, and show it to be computationally indistinguishable from an execution of the real protocol (*UC-emulation*). $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ is the random variable representing the output of environment $\mathcal{Z}$ for an execution of π in the presence of adversary $\mathcal{A}$ (conversely $\mathsf{EXEC}_{\phi,\mathcal{S},\mathcal{Z}}$ is for the execution of the ideal functionality ϕ in the presence of simulator $\mathcal{S}$).

Theorem 2.1 (UC emulation)**.** *For any PPT protocols* π, ϕ *and identity predicate* ξ*, we say that* π ξ-UC-emulates ϕ *(or simply* π *UC-emulates* ϕ *if the identity bound allows any identity) if for any PPT adversary* $\mathcal{A}$ *there exists a corresponding PPT adversary* $\mathcal{S}$ *(the simulator), such that for any balanced PPT* ξ*-identity-bounded environment* $\mathcal{Z}$*, it holds that* $\mathsf{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{EXEC}_{\phi,\mathcal{S},\mathcal{Z}}$

UC-emulation can be used to show that, if we have a protocol π that *realises* an ideal functionality $\mathcal{F}$, the security analysis of a new protocol ρ that has π as a subroutine can be carried out by replacing all of ρ's call to subroutines running π with calls to ideal functionality $\mathcal{F}$, which we denote as $\rho^{\pi\to\mathcal{F}}$. This new version of ρ is said to be in the *hybrid model*, since its ITIs interact with both other real ITIs and ideal functionalities. For the replacement to be successful, we require that any party in ρ that calls to a subroutine in π or $\mathcal{F}$ satisfies ξ and does not call instances of π and $\mathcal{F}$ in the same session (we say that the protocol ρ is (π, ϕ, ξ)-compliant). Additionally, the adversary should be able to determine whether an ITI in a certain session is part of the protocol (the protocol is subroutine exposing).

Theorem 2.2 (UC Composition Theorem)**.** *For any PPT protocols* ρ,π,ϕ *and predicate* ξ*, if* ρ *is* (π,ϕ,ξ)*-compliant,* ϕ,π *are both subroutine respecting and subroutine exposing, and* π ξ*-UC-emulates* ϕ*, then* $\rho^{\pi\rightarrow\phi}$ *UC-emulates* ρ.

Unfortunately, many interesting protocols, such as commitment schemes [79], secure two-party computation [80] or even authenticated channels [77], are not easily provable in UC in the plain model. We therefore need to add some ideal subroutine that can represent the cryptographic assumptions required as a block box ideal subroutine. The next section will discuss how hybrid functionalities that share state among sessions can also be used composably through some tweaks to the UC framework.

2.1.2.2 Globality

While UC provides a powerful paradigm for reusable cryptographic proofs, composition imposes many restrictions over the base model as outlined in Theorem 2.2. To address the limitation of the UC theorem of subroutine-respecting interactions, Canetti and Rabin [81] introduces Universal Composition with Joint-State, a new composition theorem that allows a single protocol session to be a subroutine of different protocols. This can be used for example to prove the security of different protocols that use an authenticated channel, where all sessions interacting with the same party share the signing key. This composition theorem is, however, only valid for static protocols (where the number of shared sessions is already well defined). Canetti et al. [87] formulate two new variants of Universal Composition, Extended UC and Generalised [1] UC, that allow composition when arbitrary protocol interact with the shared subroutine. The formulation of GUC has been widely used in the literature, allowing modelling of protocols that were previously impossible to prove in plain UC, such as those that provide deniability. Canetti, Shahaf, and Vald [82] later extended the GUC composition theorem to allow the replacement of global functionalities with protocols. Despite its popularity, proving security in GUC is more difficult than in the incompatible plain UC setting, as it requires arguing about all possible protocols rather than just the one being analysed. Moreover, as basic UC has received multiple updates and fixes over time, those have not percolated to the GUC formalisation, and the equivalence between GUC and the simpler EUC theorem (which most security proofs in the global setting are actually using) has been called into question due to some components of the framework being underspecified [37]. Camenisch, Drijvers, and Tackmann [71]

[1]commonly misattributed as Global UC

also show that neither the UC or GUC composition theorems allow replacing a protocol with its ideal functionality if it is both a subroutine of the protocol and one of its ideal subroutines; they therefore propose a recursive composition theorem for jointly subroutine respecting functionalities, multi-protocol UC.

Universal Composability with Global Subroutines [36] aims to rectify these issues by embedding UC emulation in the presence of a global protocol within the standard UC framework.

To achieve this, a protocol π with access to subroutine γ is replaced by a new structured protocol $\mu = M[\pi, \gamma]$, known as the *management* protocol. The management protocol is designed to be subroutine-respecting to preserve composition, while allowing the external protocol ρ to access a single instance of π and multiple of γ. μ is a shell only protocol that uses a directory ITI to redirect external communication from ρ to the appropriate machines in π or γ (and conversely to the external machine that should receive a response). The following definition roughly corresponds to the EUC formulation of global functionalities:

Definition 2.1. *For protocols* π, ϕ, γ*, we say that* π ξ-UC-emulates ϕ in the presence of (global subroutine) γ *if* $M[\pi, \gamma]$ ξ*-UC-emulates* $M[\phi, \gamma]$

As in the basic UC framework, the composition theorem follows, with some additional restrictions: π and ϕ are allowed to break their subroutine- respecting behaviour to use the global subroutine γ (we say they are γ-subroutine respecting), and γ itself does not depend on ϕ as one of its subroutines (we say that γ is ϕ-regular). These requirements allow the use of the shared state subroutine without provoking circular dependencies that would prevent a clean cut replacement [2].

Theorem 2.3 (Universal Composition with Global Subroutines). *For any subroutine-exposing protocols* ρ, ϕ, π, γ *where*

- γ *is subroutine respecting and* ϕ*-regular,*
- π, ϕ *are* γ*-subroutine respecting,*
- ρ *is* (π, ϕ, ξ)*-compliant,* $(\pi, M[\phi, \gamma], \xi)$*-compliant and* $(\pi, M[\pi, \gamma], \xi)$*-compliant;*

if π ξ*-UC-emulates* ϕ *in the presence of* γ *(per Definition 2.1), then* $\rho^{\phi \to \pi}$ *UC-emulates* ρ.

[2]This type of recursive composition is implemented in multi-protocol UC [71]; however the composition theorem of that work is not compatible with Theorem 2.2

The above theorems can be used to recover EUC statements in the literature by formulating an appropriate identity bound[3]. While most of the existing work focus on ideal functionalities as global subroutine, Badertscher, Hesse, and Zikas [34] show that UCGS does not universally preserve the composition theorem from [82] to replace the setup with a potentially interactive protocol using a different setup. In particular, when replacing a particularly weak global setup G (where adversarial capabilities are more extensive than the proposed protocol γ that realises it), the simulator S in the emulation of a G-hybrid functionality F by some protocol π might no longer be possible in the γ-hybrid world, as it can no longer use the attacks allowed by G. Their work then provides some guidelines on which global setups can be successfully replaced by a protocol. Namely, an equivalent setup (where protocol γ UC-emulates ideal functionality G, and G UC-emulates γ) can always be replaced, regardless of the context protocols which use it as a global subroutine. Additionally, replacement is possible if the simulation strategy of S either avoids using any of the adversarial capabilities of G (S is an *agnostic simulator*), or that the adversarial capabilities it does interact with will be preserved by γ (S is an *admissible* simulator).

Canetti et al. [86] later observes that the replacement statement also holds if protocol γ replaces the protocol that combines G with the simulator from the γ UC-emulates G experiment, and thus any F using that combined protocol as a global subroutine can be replaced with γ.

To conclude this section, we note that in the rest of this work, whenever an ideal functionality calls another (global) ideal subroutine (e.g. provides some input to the global subroutine on behalf of a specific party), the underlying operation relies on the intermediary dummy party convention of [36, Definition 4]. Additional remarks about our notation when we present UC protocols follows.

2.1.2.3 Notation

We now list additional convention taken by our pseudocode for the remainder of this work. This section is complementary to the earlier description of Section 1.2 and provides a more precise definition for many of the conventions used to describe UC protocols in the rest of the work. We hope our notation is generally self-explanatory, but in case of ambiguity we refer the reader to the following explanation. We might refer to UC terminology beyond what was described earlier; any such usage is self-

[3]We will show an example of this by recovering an existing global setup in section 3.4

contained to this section, but we refer to [75, Section 3.1] for additional context.

Our notation defines ITIs in terms of their behaviour when they are activated and find a new message on their input tape. We define the code executed when such a message is received as a subroutine. Some subroutines definitions are not meant to be triggered by external parties, but are simply used to extract some shared code that the ITI might need to execute multiple times. In that case, we use the keyword "run" followed by the subroutine name to denote that the same ITI is executing it. The ITI is understood to choose which subroutine to execute by pattern matching on the program definition as specified in the pseudocode, starting from the earliest subroutine definition i.e. if there are multiple commands that start with the same keyword, it will try to find the one with the correct arguments starting from the earlier definition. When the first argument is *, it is taken to be a wildcard, and when font cmd is used, it is taken to be a variable; , so any subroutine will match (and is therefore typically defined last).

Our message-passing treatment tends to stay at a higher level than the underlying UC execution. As such, we omit many details of the ITI behaviour in our protocol descriptions. We generally describe a subroutine by using the notation "*On message (*SUBROUTINENAME,*list of subroutine arguments) from party P*:" followed by high-level pseudo code for the ITI execution, in the style of an imperative programming language. This notation is short for indicating that the machine we are describing on activation reads from its input type a message of type $(P, ($SUBROUTINENAME, *list of subroutine arguments*$))$, where P is an object that contains fields $\mathsf{pid}, \mathsf{sid}$; and SUBROUTINENAME corresponds to some code in its program it can execute with the inputs from the argument list. Conversely, the notation **return** $(\text{MSG}, args)$, as part of the description of subroutine pseudocode for an ITI M, denotes the end of the execution of the current subroutine with the issuing an external write request (f, M', t, r, M, m), where destination ITI M' is the same machine from which it received input, and $m = (\text{MSG}, args)$. In this case, we always set f, the `forced-write` flag, to 1; t, the destination tape, to **subroutine-output** (unless the pseudocode describes an adversarial machine, in which case t=**backdoor**); and r, the `reveal-sender-id` flag, also set to 1. Keyword **abort**, or **return** with no arguments indicate the end of execution for the current subroutine without issuing a corresponding subroutine output message.

If M wants to issue an external write request for a destination ITI that is not the same that initiated the current subroutine execution by passing input to M, we use

"**Send** (MSG,args) **to** M'" to issue a the same message as described above, except for setting t to **input**. If the **Send** instructions is not the last one in the current subroutine description, the external write request is not issued immediately, but rather queued in the outgoing message tape for M until the end of the subroutine, or when M next relinquishes the activation token. When we use "**Send** (MSG,args) **to** M' and **receive** (MSG$'$, args$'$)", M yields activation immediately, and resumes execution the pseudocode from the same instruction when it next receives message (MSG',$args'$) on its subroutine output tape from the sender. When this happens, the ITI stores its current execution context (i.e. any intermediate computation on the work tape) somewhere in memory in a way that it can be restored when activated in this way. Between sending and receiving the response, the ITI can be activated with any other message on any tape, although if our current program can not tolerate such concurrency, the ITI might abort by checking some internal flag. If multiple outgoing messages were sent to the same M', we assume that the response includes some unique identifier to allow M to restore the correct context for which it is responding to[4]. When M issues an outgoing message, and expects the corresponding response to come from a different party, we use the keyword **await** , followed by a full description of the behaviour on next activation.

A variable assigned as part of a subroutine does not guarantee that it will be available to other subroutines, unless it is defined in the State variables table at the start of the definition. When the same program uses the same identifier across different subroutines, they are generally taken to be distinct values, especially if received as part of a message. Variables first defined within a loop or **if** branch have their scope local to that block. Protocol parameters are generally taken to be globally readable to all protocol parties and their subroutines.

2.1.3 Corruption Models

To prove the security of a protocol, it is important to state what assumptions are being made about the power of the adversary. UC represents the adversary as a polynomial probabilistic algorithm with its runtime defined by the security parameter. Additionally, some of the protocol parties are assumed to be corrupted. This implies that either one of the protocol principals wants to obtain some additional advantage over the other participants or interfere with the outcome of the protocol, or alternatively that a portion of the software they are running has been compromised.

[4]This is not a universally safe assumption to make for any UC protocol, but it is sufficently safe for the ones analysed in this work

One dimension of defining the corruption model is determining which parties are controlled by the adversary. A protocol is said to be secure against *static* corruption if the set of corrupted parties is fixed at the start of the computation. *Adaptive* corruption allows the adversary to interactively choose which parties to corrupt during the execution of the protocol. *Mobile* corruptions allow the adversary to mark a previously corrupted party as no longer corrupted. Security proofs will generally include some constrains, such that the security of the protocol holds only if the adversary has not corrupted a certain percentage of the total available parties.

The other dimension of corruption models is how corrupted parties can interfere with the protocol. We generally consider the two models of passive and active corruption. A passive adversary (also known as honest-but-curious or semi-honest) will not diverge from the protocol instructions, but is interested in finding out more information over what the protocol would normally reveal. An active adversary (also known as Byzantine) behaves arbitrarily, choosing to drop or replace protocol messages in such a way as to allow them some advantage.

UC captures different corruption models through different shells. Parties are considered honest until the adversary sends a special corruption message on an ITI backdoor tape. From that point forward, the shell implements any corruption behaviour, such as leaking the state of the internal ITI or sending malicious messages to other protocol parties. Corrupted parties are registered in a special corruption aggregation ITI, which records a partial identifier of the corrupted ITI (such that the environment is not able to distinguish between real and ideal world based on the code of corrupted machines). [75, Section 7.1] describes how to implement shells for a variety of corruption models.

Some cryptographic works consider each corrupted party to be independent, and colluding parties that share information with each other are taken as a separate case. As all corruptions in UC are issued by a single adversary (on behalf of the environment), all corrupted parties in this setting are considered to be colluding.

2.2 Trusted Execution Environments

Our work focuses on formally describing, using the tools of Provable security, and UC in particular, the usage of programmable hardware-based Trusted Execution Environments. This technology is often misunderstood, due to an abundance of confusing and ill-defined terminology, a rapidly shifting landscape of competing commercial and academic products, and designs with incompatible properties. We now attempt to give a non-comprehensive overview of the technology to ground the reader with sufficient context for situating our work into the broader field.

2.2.1 Definitions and History

There is no agreed-upon definition of Trusted Execution Environments, and how they relate to the broader class of Confidential Computing [245]. In this work, we broadly conceptualise this technology as the one demonstrated by Intel SGX and related work. This can be broadly summarised as an architectural design that allows, through specific changes to the memory architecture and the use of cryptographic techniques, to run arbitrary computer programs on an otherwise untrusted machine, with no loss of integrity and confidentiality (for code and / or data), and prove to a remote verifier that the TEE is running such program with the adequate protections (a process known as *remote attestation*). The execution context usually takes one of two styles: (i) process-based TEEs, which run a minimal program with access to shared facilities outside the protection mechanism such as the operating system kernel; and (ii) virtualisation-based, where the entirety of a virtualised OS is protected. While (i) attempts to minimise the Trusted Computing Base (TCB i.e. the surface area of system components whose failure might lead to a compromise), (ii) is increasingly favoured by TEE vendors due to the greater ease of the development, despite the larger attack surface. In general, the threat model addressed by TEEs is the protection of a program, running as an *enclave* on a device along other malicious privileged software. The operator of the machine running the TEE might also be considered adversarial. Most TEE designs however do not provide protection against physical attacks or guarantee availability (any additional component to obtain guarantees of the latter can be considered part of a Reliable Computing Base [234]).

Weinhold et al. [295] identify the crucial definitional component of TEEs as the ability to provide continuing remote attestation of an enclave over time. They further break down TEEs into the following constituent elements:

- An execution context for arbitrary programs
- The ability to collect cryptographic evidence (known as *measurement*) of the program being executed
- A hardware Root of Trust to assert the veracity of the measurement

- Memory isolation (at rest and during execution) from other programs, potentially at high privilege level, running on the same machine
- The ability to run multiple execution contexts on the same machine
- A communication channel between the execution context and untrusted sources such as I/O devices

Sardar, Fossati, and Frost [247] provide a characterisation of remote and local attestation in the setting of TEEs, extending the standards-driven definition of the IETF RATS (Remote ATtestation ProcedureS) working group [56]. Attestation is a protocol between three parties: an attester, a verifier and a relaying party (the last two often being the same entity). There are four phases to an attestation protocol, each executed with increasing order of frequency. The first phase, *provisioning*, takes place once in the lifetime of a TEE-enabled machine (such as the manufacturing facility) to hard-code platform specific identity values and secrets. The *initialisation* phase derives additional secrets during the platform boot sequence. The third phase, *attestation* itself, can be broken down into subphases: first the attester generates evidence, then the verifier runs an evidence appraisal algorithm, and afterwards the relying party runs an attestation appraisal algorithm. Finally, depending on the success of the appraisal process, the relying party makes a *trust decision* on whether to carry out any trustworthy operation with the attester.

Feng et al. [125] identify the parties of typical confidential computing applications as *Service Provider, Data Owner, Code Owner,* and *Client*. The Client receives a verifiable output of running a program designed by the Code Owner, on the Service Provider's infrastructure, with the Data Owner providing the input. While some of these parties can correspond, Confidential Computing applications should allow all four parties to be distinct entities that are mutually untrusted.

Precursor technologies The ideas that would lead to the creation of Trusted Execution Environments long predate the technology's introduction in it's current form, and have led to the development of several earlier designs, often broadly described as *protection modules*.

Starting in the late 1970s, computer manufacturers such as IBM [28] developed multiple generations of purpose-built hardware to isolate cryptographic key material and accelerate cryptographic operations. We generally refer to this kind of technologies as *Hardware Security Modules (HSM)*, or *security co-processors*. In 1980, Best

[52] proposed the idea of *crypto-microprocessors* to prevent the unauthorised redistribution of programs or the sensitive data they held by distributing them in encrypted form and storing the key in a protected section of the CPU. By 1997, Kuhn [170] had further refined this design to allow mutually untrusted encrypted programs on an untrusted operating system to run in encrypted form on a generic CPU with a modified memory model. Concurrently, Arbaugh, Farber, and Smith [25] proposed underpinning the security of an operating system boot process by introducing a verification chain for each subsequent component of the system as it is loaded.

The combination of these ideas was a core part of the plans of the Trusted Computing Platform Alliance (later Trusted Computing Group), an industry consortium formed by Microsoft, Intel, IBM, HP and AMD in 1999. The group promoted the deployment of Trusted Computing (sometimes trustworthy or safer computing), a combination of hardware and software feature that would allow consumer operating systems to prove they were running "untampered software". While the stated goal was the development of a more secure version of the Windows operating system that would rely on Trusted Computing (the Next-Generation Secure Computing Base, code-named Palladium), it would also enable previously inaccessible measures such as Digital Rights Management and preventing the execution of unregistered software (such as pirated programs)[19]. The consortium received much criticism for how their efforts would limit the ability of computer owners to run what they wanted on their own hardware [5], and while initial efforts were originally scaled back, an outcome of the group was the standardisation of the *Trusted Platform Module (TPM)*. TPMs combine the secure cryptographic functions of an HSM with the ability to securely store encryption keys for memory operations (sealing) with complex policies, and the addition of anonymous remote attestation capabilities to prove that firmware running on the TPM and the host platform in general has not been tampered with. TPMs can use their measurement capability to act as the root of trust for a verified boot chain, where each level of the boot sequence can measure the next, and decryption of memory can be pinned to an accepted sequence of measurements [227]. This use of TPMs as a trusted root of trust was commercialised by the release of Intel Trusted eXecution Technology (TXT), a CPU feature that allowed proving that a Operating System kernel or virtual machine hypervisor was loaded in an environment where all of the required underlying components were running genuine untampered firmware, thus providing guarantees of launch-time security, but none over runtime security (e.g. any attacks

[5]as described in our Introduction, and further documented by Anderson [18]

due to a buffer overflow in the trusted firmware, which includes the complex hypervisor cores)[299]. Thus, from the desire of minimising the *Trusted Computing Base* (the set of trusted components necessary to ensure the secure operation of a system) Intel Software Guard eXtension (SGX) and *Trusted Execution Environments (TEEs)* were born. For a technical overview to the development of the above technologies and competing non-Intel designs, we refer the reader to [104, Chapter 4].

TEEs today Without fully jumping into the detail of TEE architectures, which we do in the next section, the core aspect of the technology is the ability of running trusted processes along with untrusted ones, including an untrusted operating system or hypervisor. Competitors to SGX for each different CPU architecture have been proposed, all with slightly different guarantees and components. SGX and TEEs have attracted much more attention than their predecessor technologies, both in terms of their proposed use cases and the numerous attacks discovered. While receiving much attention from the academic research community and the press, the same can not be said for application deployments. Both Intel and ARM, the biggest CPU architecture vendors, have struggled to persuade developers to write custom software that would run on their TEEs. Applications can not be trivially ported to enjoy the security benefits of the technology, and particular care has to be taken to ensure that sensitive data is not accidentally accessible from untrusted processes. To facilitate development of TEE applications, a few projects (e.g. [49, 27, 281]) embed a Library OS (in this context a re-implementation of normally untrusted system calls) within the enclave, to allow porting existing software without modifications. While a growing number of cloud computing vendors have explored the adoption of TEEs to secure sensitive workload of their customers, SGX has been discontinued from consumer CPUs and relegated to the server market. A new successor technology, Intel Trusted Domain eXtension (TDX), has been introduced. TDX provides similar guarantees to SGX, but for virtual machines, thus without requiring software changes to applications, at the cost of an increased trusted computing base.

Despite the longer timelines of hardware development compared to software, the landscape of TEEs is moving relatively fast: SGX was first launched in 2015, and applications of Confidential Computing are an active area of work in both industry and academia. Several organisations and standard bodies are actively working on providing industry standards for TEEs and hardware security in general; we note

- The U.S. Federal Information Processing Standards 140 series (FIPS-140) issued by NIST [256] specifies the requirement for cryptographic modules (such as HSMs and TPMs)

- The Trusted Computing Group offers certifications for TPMs and Attestation protocols
- The GlobalPlatform TEE committee standardises HSMs (Secure Elements) and TEEs for embedded platform, including ARM TrustZone
- The Confidential Computing Consortium, part of the Linux foundation, stewards the development of standards and open source projects to enable TEE-driven confidential computing through a variety of projects at different levels of the TEE stack
- The IETF TEEP (Trusted Execution Environment Provisioning) and RATS (Remote ATtestation ProcedureS) working groups are developing standardised protocols to deploy applications to TEEs [282] and for remote attestation [56]

2.2.2 Architecture

We now detail some of the fundamental architectural components of Intel SGX and its attestation mechanism to provide an example of how TEEs can be implemented. This is a high level overview, and we refer an interested reader to Costan and Devadas [104] for additional details, and to McKeen et al. [208] for an overview of the changes since introduced by the second major version of SGX (commonly referred to as SGX2). We note that much of the functionality of SGX and TPMs are interlinked through common usage of Intel's Management Engine [239], a technology that has proven to be controversial for not allowing computer owners to inspect or modify its behaviours [89] and increasing the attack surface of general purpose computers [241]. Perhaps due to the controversial nature of the Management Engine, its role in SGX is not mentioned often in technical documentation.

Intel SGX provides process-based isolation, by allowing developers to write custom programs called *enclaves*, whose environment is isolated by the rest of the software running on the processor, including the operating system. The Trusted Computing Base of SGX is limited to microcode instructions that allow enclave execution, and some architectural enclaves designed by Intel. The adversary SGX defends against is one that controls all software on the host, but can't undertake sophisticated physical attacks. SGX supports running multiple concurrent enclaves, including multi-threading.

The security of SGX relies on a subset of memory on the CPU reserved for storage of enclave programs and their data, and whose access is protected from all other software (disjoint-memory assumption [175]). While the memory is still managed by the same mechanisms that allocate untrusted programs, SGX performs certain checks that prevent invalid usage, such as preventing the same physical memory region from being allocated to two different enclaves. Within the protected memory region, SGX enclaves are reserved a custom page cache. Any pages evicted from the cache is stored to disk in encrypted form by using a symmetric “sealing” key.

The virtual memory for an enclave process is mapped to both a secure memory region, and insecure memory that belongs to the host process that initialised the enclave. The former should be used to store any sensitive data, and has to fulfil certain safety criteria, whereas the latter is used as an interface with non-trusted programs. Since the loading of memory is performed by the untrusted firmware, enclave developers are allowed to tag memory section with permissions, which the trusted components ensure are being respected. The secure memory region also includes space to save a running enclave’s state when the (untrusted) firmware triggers a hardware exception.

For the purposes of untrusted program, an enclave can be used as the equivalent to a dynamically loaded library. A host application instructs the CPU to create a new enclave, which initialises the appropriate memory region and enclave metadata. The host can then load code into the memory region, and initialise the enclave using a system-provided *Launch Enclave* (code can only be loaded into uninitialised enclaves). Once initialised, the enclave’s code is executed by switching a thread to a special enclave mode, which can to access the protected memory region without increasing process privileges. Enclave mode can be exited synchronously (with the enclave program yielding back control) or asynchronously (due to a hardware exception), in which case an exception handler stores the enclave’s state into trusted memory, erases all memory for the execution context, and yields to the untrusted system software to handle the exception outside of enclave mode. A similar mechanism is provided to resume the enclave after such an exception. Additionally, the host can also teardown an enclave that is not currently running, permanently erasing all of its memory.

Attestation SGX implements an attestation scheme similar to the one enabled by TPMs, but with the ability to limit the scope of the attestation to a specific enclave rather than the entire system.

A hash of an enclave’s state at initialisation (including any code and data loaded by the host, and any CPU features required by the enclave) is known as a measurement. Each enclave developer can issue a certificate to sign the measurement with a key tied to their identity (this is also useful to allow an updated enclave to receive any secrets held by the old version).

An enclave can prove it is running a certain program to a local verifier (e.g. another enclave running on the same machine) through a process known as local attestation. Local attestation produces a Message Authentication Code over the signed enclave measurement and some arbitrary data chosen by the enclave.

A special purpose architectural enclave, known as the Quoting Enclave, is used to provide attestation to a remote verifier. The quoting enclave is provisioned (through a Provisioning Enclave) with an attestation key that depends on the current firmware version for SGX, and is stored encrypted in non-volatile memory. Later, the quoting enclave can receive a local attestation from any enclave and, if valid, replace the MAC value with a signature produced by the attestation key and send the resulting quote to the verifier.

There are several flavours of attestation protocols that have been used with SGX. The original Enhanced Privacy ID (EPID) scheme uses a group signature to provide privacy preserving attestation. The verifier can forward a received quote to the Intel Attestation Service (IAS), which verifies that it is signed by a valid up-to-date CPU on behalf of the verifier. The signature schemes ensures that the attestation service can't verify the identity of the quoted enclave, or link any quotes to each other. It is up to the Intel Attestation Service to maintain a revocation list for any compromised signatures. Intel has officially announced the End-of-Life date for EPID-based Intel Attestation Service, replacing it with the new Intel Trust Authority service which implements RATS attestation. Another alternative to the Intel Attestation Service is the Data Center Attestation Primitives (DCAP), which allows decentralising attestation to any provider. DCAP uses ECDSA signatures and does not support the anonymous attestation feature of EPID. For the remainder of this work, we consider attestation to support the anonymity feature.

2.2.2.1 Alternative Designs

While the above architectural overview should give the reader an understanding of some of the design decisions in SGX, there are many different implementations of TEEs, each with its own architecture and security guarantees. As a commercially successful example, ARM TrustZone [26] is a proprietary extension of the ARM architecture that allows CPUs to execute secure applications. It provides physical separation between the memory and execution contexts of a “secure world” and “normal world”, both running within the same processor. Despite predating Intel SGX, the number of applications that have been developed in TrustZone has been limited, due to the relatively more closed nature of its developer toolkit, as well as an inflexible memory model and performance issues. More recently, the new Confidential Computing Architecture in ARMv9 CPUs adds secure virtualisation capabilities (akin to Intel TDX) and additional security features [155].

The open ISA RISC-V has also seen several proposals to add (mostly) process-based TEE extensions: MIT Sanctum [105], Keystone [176], Multizone [138], Citadel [123], Mi6 [60], CURE [39], and HECTOR-V [215].

Other commercial based TEE vendors include AMD (SEV-PNP), IBM (Z Secure Execution), Amazon (AWS Nitro). Schneider et al. [253] provide a more comprehensive list of commercial and academic designs.

Heterogeneous TEEs Both Intel SGX and competitors are strongly tied to the CPU architecture and its instruction set (ISA). Weinhold et al. [295] propose an ISA-independent TEE design that allows a modular choice of implementations of some TEE components. Ferraiuolo et al. [126] propose Komodo, a software-defined enclave architecture instantiated under ARM TrustZone

More generally, we use the term *heterogeneous TEE* to refer to a hybrid TEE architecture that incorporates non-CPU components in the TCB. The NVIDIA H100 GPUs includes a confidential computing mode [117], while HIX [159] and Graviton [290] enable GPU-based TEEs, through minimal hardware changes to the memory controller and by loading the GPU drivers in a CPU enclave, respectively. While not designing a novel heterogeneous TEE architecture, Slalom [277] outlines how the separation of trusted and untrusted components of a program might be used to provide performant private and trustworthy GPU acceleration of certain operations along with the CPU TEE. Similarly, Ghosh et al. [140] and Xia et al. [302] both provide secure mechanisms for SGX to offload work to a Field-Programmable Gate Array (FPGA) accelerator, and Zhu et al. [313] and Schneider et al. [252] provide similar mechanisms for constructing enclaves that can safely rely on different types of accelerators and I/O devices. Other FPGA-based TEE designs have also been proposed by [222, 232, 308], and [115, 161, 283] all propose mechanism for Arm TEE technologies to securely access GPUs or FPGAs.

Other theoretical designs have been proposed to further develop the capabilities of TEEs. Li, Xia, and Chen [183] introduce the concept of plugin-enclaves, smaller attested components that can be mapped into multiple larger enclave's memory to minimise resource usage. Similarly, Yu et al. [307] designs a system to selectively share memory between different enclaves; Park et al. [226] proposes the concept of nested enclaves to enable fine-grained hierarchical levels of protection to mutually distrusting enclaves that might need to share resources; and Zhao et al. [311] proposes multi-layer permission systems for enclave execution and attestation to increase performance in the enclave creation phase.

2.2.3 Applications and Attacks

Applications Although the commercial aspirations of TEE vendors might not have been met by real world adoption, there is a wealth of research into the potential use of TEEs and a growing ecosystem of software to enable it.

Li [184] attempts to keep an up-to-date index of both academic papers and open source software that relies on TEE technologies. Recent academic work [298, 225] has surveyed the literature of TEE applications, but this kind of efforts are doomed at becoming quickly out of date. With that in mind, we do not attempt to replicate the work of other surveys. Instead, we note the broad categories in which they classify previous work.

Both surveys classify previous work in the TEE literature along two criteria: what security goals they try to achieve, and the context they can be used in. The following security properties (which are further devided into subproperties) are present in both works: Privacy, Integrity, and Confidentiality. Will and Maziero [298] further adds the categories of Enclave Management and Authentication, and Paju et al. [225] identifies Cryptography, Attestation, Blockchain and Decentralisation as distinct categories. Will and Maziero [298] distinguishes TEE applications into a local and distributed context. Local TEE applications are used to strengthen the guarantees of Operating Systems, provide local runtime security to unmodified applications, and develop purpose-built secure applications. Distributed applications include implementing network infrastructure, distributed application runtime, data storage and sharing, data analytics and Internet of Things applications. Paju et al. [225] find more than a 100 application use cases, divided in the categories of Data Analytics, Cloud Computing, Access Control, Data Protection, Online Payments, Memory Pro- tection, Attestation tools, secure Storage, Network Security, secure Channels, Content Sharing, secure Code Offloading, Smart Contracts, Computer Games, Hardware Accelerators, Formal Methods, Medical Data, secure System Logging, Web Search, Data trading, and Digital Contracts.

Attacks As a primarily security driven technology, a crucial concern towards the adoption of TEEs is how well they are able to protect their programs. There are clear causes of concerns around the effectiveness of TEE architectures, given the number of attacks that have been discovered since their introduction.

Feng et al. [125] and Schaik et al. [251] categorises the literature of attack vectors on TEEs into some broad mechanisms: fault injection, transient and speculative execution, side channel attacks, system and software vulnerabilities (which include OS privilege escalation, memory corruption etc) The possible consequence of these kinds of attacks recorded in the literature include: loss of confidentiality of enclave code and data; loss of integrity due to tampering with enclave code, faking attestation, or lack of memory safety guarantees; breaking isolation from other processes; and reverse engineering.

These surveys also discuss mitigation techniques. Some defenses are provided by the TEE vendor, whereas some attacks can only be addressed by defensive programming mitigations, which require a library or application update. Other attacks could be prevented by shifting the boundary between TCB and untrusted code for a specific TEE architecture

Schaik et al. [251] further notes that Intel issues vulnerability mitigations through microcode updates bundled through BIOS update for each motherboard. Due to the inherent risk of damaging a machine with such an update, a high number of CPUs in the wild do not apply vendor patches, leaving the attestation relying party with the choice of reducing the pool of attested machines they are willing to interact, or increase the probability that they might be affected by a vulnerability that has been patched by the vendor.

More recent surveys analyse the class of power management related attacks in detail Gonidec et al. [145], and classify how the design choices of a TEE architecture might lead to specific types of attacks [181].

We now examine in detail a specific class of memory guarantees attacks (and their mitigations) that will be relevant later in this work.

2.2.3.1 State Continuity

One important property of programs executed on an untrusted host through trusted execution is State Continuity, the ability of the trusted program to be run without the adversary interfering with the scheduling of operations it performs. If state continuity is not guaranteed, an enclave that encodes state is not truly protected from the host. To allow an enclave program from resuming after an interruption, most TEE platforms allow enclave memory to be stored on untrusted non-volatile memory as an encrypted object, with the decryption key only available to the original enclave. While this mechanism is intended to protect confidentiality, the untrusted nature of the storage medium can lead to state continuity attacks.

Besides the potential for denial of service, there are two types of tightly correlated attacks that affect state continuity: rollback attacks and forking attacks.

A *rollback attack* allows the adversary to rewind the internal state of an enclave to a previous one by interrupting it and restarting it with an old copy of its internal state (*stale response*), by forging a new memory state (*synthesized request*), or by applying legitimate state updates from the TEE multiple times (*replay*)[21].

Forking attacks allow the adversary to maintain multiple copies of the same enclave, either through cloning [65] or by repeated applications of rollbacks, allowing the adversary to present inconsistent states to anyone communicating with the enclave.

We also note that Jangid et al. [160] considers a broader notion of state continuity that also includes the safety of global variables in multi-threaded enclave programs, an important consideration that is beyond the scope of our work and indeed most of the existing literature discussed below, which seemingly holds an implicit assumption of thread-safe execution.

State continuity in earlier protection modules The study of state continuity precedes trusted execution, and previous works in the context of protected modules attempted to address these attacks. While some work proposed to increase the TCB to include a wider I/O interface that includes memory storage, a simpler solution relies on checkpointing memory dumps with the value of a trusted monotonic counter. This can be implemented through using a TPM [179] or NVRAM as non-volatile memory, or as a service provided by the TEE platform [90] [6]. A monotonic counter should be *consistent*, when its value is always equal to the number of times the counter was

[6] although the counter service is not supported on all versions and operating system combinations for which the Intel SGX SDK is available [267]

incremented, and *safe*, where an attempt to reset the counter can only erase it rather than reducing the value [197]. Monotonic counters are not sufficient to protect against forking attacks if the enclave can also be cloned [65].

Parno et al. [228] shows that state continuity can be guaranteed by achieving continuous storage. In practice, however, these techniques have severe limitations: increased latency, which makes it unsuitable for any application setting requiring high throughput; and a finite number of updates for the lifetime of the hardware resources. Thus, a local adversary can readily defeat these security measures by effectively conducting a denial of service attack on its own hardware. The classic approach of 2-step commitment is also not suitable for ensuring termination in the context of protected modules [228], as the first phase necessarily reveals the content of the state. Encrypting the commitment information can result in leaking details about the state or output through side channels, allowing the adversary to abort the commitment and therefore rollback.

Additionally, the logic of counter increments and verification can lead to subtle bugs that could jeopardise integrity or crash resilience. Matetic et al. [200] distinguishes between two common techniques, *inc-then-store* and *store-then-inc* (also referred to in other works as *increment-and-persist* or *execute-then-record*, and *persist-and-increment* or *record-then-execute*, respectively). As the name implies, the former triggers an update to the monotonic counter and then stores the new value in untrusted memory along with the state, while the latter commits state to memory before updating the counter. If the system experiences a crash between the two operations, an inc-then-store application will not be able to recover, because the monotonic counter has been increased but no state has been stored for that value by the application. Additionally, inc-then-store systems are susceptible to side-channel attacks, where the adversary might interrupt the computation and reset the state before the counter is updated multiple times. On the other hand, a crash between storage and counter increase in a store-then-inc application would allow the adversary to replace the previous state. Jangid et al. [160] shows an attack to a protocol that leverages monotonic counters to ensure that a certain amount has passed before the enclave can issue a certificate. Since the vulnerable version of the protocol increases the monotonic counter after the certificate has been issued, an attacker can block the enclave between these two steps and use another copy of the enclave to issue a second certificate, bypassing the check. They address this by incrementing the counter (instead of only reading it) before the untrusted timer data is unsealed, essentially moving the protocol to inc-then-store..

Protecting TEEs from State Continuity attacks In the context of TEEs such as SGX, a number of proposals to handle state continuity attacks involve the usage of architectural modifications. Strackx, Jacobs, and Piessens [262] introduce ICE, a novel rollback protected sealing service against an adversary who controls the CPU's power supply. Through hardware modifications, ICE detects when the platform is being powered down and copies the enclave's state into the TPM NVRAM. Unlike the naive monotonic counter approach, NVRAM access is only required at boot and during an attack, improving the durability of the service (provided the adversary does not sabotage its own hardware). Strackx and Piessens [263] reduces the stress on TPM hardware by implementing an update service that relies on a single bit flip per-update for deterministic enclaves. Bailleu et al. [43] and Gregor et al. [147] propose a primitive called Asynchronous trusted monotonic counter, which relies on the client of their Key Value store system to buffer update values in their own memory as to minimise the number of counter updates and the window of possible attacks between synchronisation points. To detect forking attacks, Briongos et al. [65] establishes a protocol based on cache contention that allows enclaves to detect whether a clone is running on the same host.

Besides the above proposals relying on architectural changes, another popular approach to preserve state continuity is to use a distributed protocol. While of course including a remote counter storage service within the TCB would allow for a simpler protocol, most of these work assume partially corrupted remote parties.

ADAM-CS (Martin et al. [197]) proposes a distributed network of nodes that use TPM counters to prevent rollback protection using a store-then-inc technique, supported by local virtual counters. Their service does not prevent all rollback attacks but tries to minimise the amount of time in which they are possible.

The *Lightweight Collective Memory (LCM)* protocol by Brandenburger et al. [62] guarantees both fork-linearisability and operational stability of a service running in an enclave on an untrusted server. The former refers to the notion that the untrusted server can not maintain different views of its clients' transaction history for different subset of clients, even if there is no client to client communication; the latter property ensures that each enclave is aware of whether their state updates has been successfully distributed between sufficient clients. LCM allows a group of clients interacting with the same enclave on a remote server to detect whether the server has mounted a rollback or forking attack on the enclave. Additionally LCM allows enclave migration to a different host.

ROTE (Matetic et al. [200]) is a distributed protocol that allows a honest majority set of TEE-equipped hosts to maintain a distributed monotonic counter platform within the context of adversarially controlled network and platform restarts. It provides all-or-nothing rollback protection (the adversary can only conduct a rollback attack if it resets every party in the protection ring) for a fixed set of parties. All parties are assumed to be honest during the setup phase, which includes a trusted third party. Rollback protection is guaranteed as long as the number of unavailable or malicious parties (including those that can tamper with enclave outputs) is always less than majority. Niu et al. [220] claims that the ROTE restart protocol is susceptible to an attack that creates a parallel protection groups, as enclaves do not check whether they are the latest running copy on that platform in the view of other participants. Similarly Briongos et al. [65] notes that if the ROTE protection group members are corrupted just after setup, they are able to run parallel protection groups by cloning the enclaves on a sufficient number of platforms.

The *Nimble* (Angel et al. [21]) protocol uses preexisting distributed crash-fault tolerant storage service to guarantee rollback protection for an application running on an enclave. A set of TEE endorsers, who do not need to run a replication protocol between them, attest to the client that they are receiving a fresh value from storage. Nimble satisfies safety as long as a quorum of endorsers is never rolled back, guaranteeing linearisability of the storage service under a malicious provider. It also proposes a reconfiguration protocol to replace the set of endorsers dynamically.

The *Enclave-Ledger Interaction (ELI)* protocol (Kaptchuk, Green, and Miers [163]) allows an enclave state to be safely outsourced to an append-only secure ledger (such as a blockchain), as well as the ability to trigger enclave execution based on externals messages published in the ledger. The protocol essentially describes a technique to construct stateful and randomised enclave programs with rollback protection from a stateless deterministic enclave and a ledger resource.

Similarly, Brandenburger et al. [63] leverages a blockchain with *final* consensus mechanisms (where it is impossible for ledger transactions to be reverted due to a fork) to design a private smart contract system that uses TEEs and is not susceptible to state continuity attacks due to using the ledger as program state.

Unlike the previous two system, *NARRATOR* (Niu et al. [220]) uses the blockchain during its initialisation phase only (rather than recording each state transaction on-chain) to bootstrap a distributed system that can ensure rollback protection for client TEEs. Their design is similar to that of ROTE, but provides a store-then-inc counter (since they assume deterministic programs), and replaces the trusted initialisation phase with one based on storing each node's unique platform identifiers on the blockchain, to avoid the co-existence of multiple nodes on the same platform. Its restart protocol addresses some shortcomings of ROTE, and setup relies on a single trusted TEE leader rather than honest parties. The authors note that to ensure state continuity, a weaker property than consensus is required, as the order of operations of enclaves on the same platform is not always important.

This is further developed in Dinis, Druschel, and Rodrigues [119], which proposes a new model of *Restart-Rollback Fault Tolerance* to capture the integrity properties of an enclave. A RR Fault Tolerant protocol allows nodes to become unavailable (as in Crash-Fault Tolerance) and to rollback their local enclave to a previous state, while maintaining the expected non-byzantine behaviour of enclaves. It provides a technique to adapt CFT protocols to become RRFT by using an adaptive per-node quorum with no significant performance loss. The newly RRFT protocols are then used to build a replicated metadata service for cloud comping (TEEMS) with better performance than those that rely on Byzantine-Fault Tolerant protocols.

While RRFT might be sufficient to protect against rollback attacks in the consensus setting, there are some indications that it might actually be a lower bound. Despite several attempts to use protected modules [99, 179, 162, 289, 305] and TEEs in particular [51, 42, 111, 137, 240, 193] to ensure that CFT protocols can be run without byzantine behaviour (*hybrid BFT* protocols), Gupta et al. [150] and Wang et al. [294] independently show that all such systems' safety guarantees are undermined by rollback attacks. The latter proposes as a solution *ENGRAFT*, a hybrid BFT protocol that prevents rollback attacks through TIKZ, a modified version of ROTE running on the same nodes as ENGRAFT, whose quorum must satisfy the same properties as ROTE.

2.2.4 Modelling

Another issue that might have hurt the adoption of TEEs, especially in the light of the many vulnerabilities discovered, is that most commercial vendors' solutions are underdocumented (or use machine-generated, unreadable specifications) and do not clearly specify the security assumptions they satisfy. As a consequence, much effort has been spent by the academic community to understand the behaviour of these technology through reverse engineering or parsing long specifications. In an environment where it is difficult to assertain the security properties from the vendor information, it is

	Guarantees	Counter	Ledger	Quorum	Setup	Reconf
ADAM-CS [197]	Minimise rb window	StI	N	0 or N-1 counters	TPM	N
LCM [62]	Fork-Linearisability	N/A	N	N/2 clients	Trusted admin	T
ROTE [200]	BFT	ItS	N	$\frac{N}{2}$,u+f+1	Trusted	N
Nimble [21]	CFT safety	ItS	Implements	N/2	Honest, TTP	Y
ELI [163]	Stateful programs	ItS	Storage	ledger	untrusted	N/A
Hyperledger [63]	Final Consensus	N	Storage	ledger	Trusted	N
NARRATOR [220]	CFT safety+liveness	StI	Idenitities	$\frac{N}{2}+1$	Untrusted	N
TEEMS [119]	RRFT	ItS	N	Dynamic	Untrusted	N

Table 2.1: Classification of rollback protection mechanisms. The Guarantees column summarises the protocol's goals. Counter indicates whether the protocol provides as Store-than-Inc (StI) or Inc-then-Store (ItS) counter. Ledger shows if and how the protocol makes use of a trusted ledger, such as a blockchain. Quorum indicates how many honest parties are required for the protocol to succeed. Setup recaps the setup assumptions for the protocol. Reconf indicates whether it is possible to migrate the enclave or modify the set of participants (T indicates that reconfiguration is possible through a Trusted admin).

useful to develop specifications against which it is possible to compare the behaviour of an implementation. Indeed, Pinto Gouveia et al. [234] argues that, for all its limit, constructing a reliable TEE will inevitably involve somew level of formal verification.

The majority of work that examines TEEs through formal verification tools, such as the ones described earlier in Section 2.1, spans well beyond the cryptographic community. As such, we now give a brief survey of work in this area we are aware of, both in the realm of traditional proofs and automated model-checking. We classify these works into two broad categories: works whose aim is the verification of a TEE architecture (or a custom extension), and those concerned with the security of applications running on TEEs. We do not survey the rich literature of formalising secure hardware beyond the scope of TEEs.

Validating TEE designs Companies that develop CPU architectures are generally not well known for the transparency of the development process, and this extends to their TEE products. Some of them have however attempted at publishing some of the formal guarantees for their designs. In [178, 208], Intel researchers address the issue of concurrency issues in SGX and SGX2, respectively, by providing a model checker to automatically prove the property of linearisability for its instruction. Fox et al. [129] combine a variety of formal methods to evaluate the design, implementation, and runtime guarantees of a trusted components within ARM CCA.

Independently from the vendors' efforts, a line of research by Sardar et al provides proofs of security in the symbolic model using the ProVerif tool for various TEE attestation protocols including EPID [249], Intel DCAP [244], SCONE attestation [246], Intel TDX [248] and ARM CCA [250] attestation.

Some academic designs for TEEs and TEE augmentations employ formal methods to provide some guarantees of their safety. For example, the design and implementation of Komodo [126] are validated through Dafny, through a custom model of the ARM architecture. Antonino, Derek, and Wołoszyn [22] formulate a technique to provide additional guarantees to AMD SEV virtual machines through an Intel SGX enclave, and prove their design satisfies its security guarantees using Tamarin. Crone [106] constructs a side-channel resilient TEE using the formally verified seL4 microkernel [168]. Busi et al. [70] provides a formal model of the Sancus TEE [221]. Interestingly, their model is shown to not adequately model some concrete attacks on the implementation by Bognár, Bulck, and Piessens [58], highlighting a limitation of formal methods' inherent abstraction of a system. To address this result, Busi, Focardi, and Luccio [69] develop a tool that generates a model through observing direct interaction with Sancus programs, and provide guarantees of noninterference (or find an attack).

Following our previous survey of state continuity attacks in Section 2.2.3.1, we highlight some relevant works that rely on formal method techniques. [21] machine checks the proof of their rollback protection protocol using Dafny/IronFleet, and Ahman et al. [11] provides a verifiable implementation of the Ariadne [263] rollback protection protocol in F$\star$ [160] studies the behaviour of TEEs in the presence of state continuity attacks by treating the TCB and enclave programs as the honest parties in a distributed system in the Dolev-Yao model, with the untrusted code as the malicious parties.

Security of protocols that rely on TEEs We now list, in no particular order, some of the works we are aware of that attempts to formally prove the security of applications and protocols that rely on TEEs.

- Sinha et al. [259] and Sun and Lei [266] propose an Intel SGX application development technique, and an ARM TrustZone variant, respectively, where the trusted component is narrowed down to a well specified interface between the secure world and the operating system through a security monitor, whose safety is easy to formalise and verify.

- Xu et al. [304] propose a model in the Tamarin prover to automate security proofs of protocols that rely on TEEs.

- Sinha et al. [260] propose a verification toolkit for assessing whether an application developed for SGX fulfills the claimed confidentiality and integrity guarantees based on its usage of the SDK, providing one of the first machine-checkable models of SGX APIs.
- Dokmai et al. [121] reduce leakage resilience their TEE-enabled application to the safety of the type-system of the Rust programming language
- Antonino, Woloszyn, and Roscoe [23] formulate a new notion of correctness for enclave execution, and provides a program analysis tool to ensure that the required conditions are met.
- Vukotic, Rahli, and Esteves-Veríssimo [292] propose a new language to assert safety property of hybrid fault tolerance protocols (where some of the replicas, running on SGX, are more trustworthy than others).
- Subramanyan et al. [265] propose the Trusted Abstract Platform, a formal model of TEEs that captures security against local privileged processes and remote attestation. TAP is shown to capture both the Intel SGX and MIT Sanctum architectures with a machine checked proof of refinement. Lee et al. [175] later extend the TAP model to capture memory sharing capabilities between enclaves, and Gaddamadugu [136] provides a machine checked-proof using this variant.
- Fisch et al. [127] define a game-based model of SGX to prove the security of a protocol that implements Functional Encryption.
- Barbosa et al. [47] are concerned with the problem of proving the security of TEE programs in a composable way, and provide a custom game-based security model that can capture TEE execution and attestation. Jacomme, Kremer, and Scerri [157] provides a SAPIC/Tamarin machine-checked simplification of this model, and Bahmani et al. [40] later realise a secure MPC protocol using the model in a simulation-based proof.
- Lu et al. [190] provides a simulation based security definition for a TEE with a partial corruption model.
- Pass, Shi, and Tramèr [230] gives a definition of TEEs under the Universal Composability setting (which we described in Section 2.1.2.1).

Given the desirable composition guarantees of UC, we choose the latter model to conduct the security analyses in this work. We now provide a detailed overview of their formalisation.

2.2.4.1 TEEs under Universal Composability

While various works exist to model HSM-like functionality in UC (e.g. see [164]), and some initial work has been proposed by Canetti et al. [88] to give a UC treatment of validating the security guarantees of generic hardware constructions (including protecting against side-channel attacks), Pass, Shi, and Tramèr [230] provide the first UC formulation of TEEs. Their G_{att} functionality (fully reproduced below in Figure 2.1) is a generalised TEE model that aims to capture architecture independent properties. It distills the essence of TEEs into attested execution i.e. evaluation of a program with associated proof of execution. on capturing the concept of *attested execution* in a general manner, removing implementation details. G_{att} lets a pre-established set of parties, with local access to a TEE, install and execute arbitrary enclave programs, which produce anonymous attestation signature over the program output and enclave metadata. While the environment is able to install their own programs through a corrupted party and verify the authenticity of an attested output, they learn nothing about the internal state of an enclave or the identity of the party executing that program. Any implementation details of the trusted hardware or concrete attestation protocol are abstracted away from the attested execution formalism. Through its simple signature mechanism, which collapses local and remote attestation into a single operation, G_{att} incorporates both the roles of attester and verifier into one setup functionality.

The functionality is parameterised with a signature scheme and a registry to capture all platforms with a TEE. The functionality in Figure 2.1 diverges from the original one in that we let vk be a global variable, accessible by enclave programs as $G_{\mathsf{att}}.\mathsf{vk}$. This allows us to use G_{att} for protocols where the enclave program does not trust the caller to its procedures to pass genuine inputs, making it necessary to conduct the verification of attestation from within the enclave.

The INSTALL and RESUME subroutines can only be triggered by parties who have access to TEE hardware (a static set defined as functionality parameter reg); but any party can obtain the verification key. On enclave installation, its memory contents are initialised by the specification of its code; this initial memory state is represented by symbol $\emptyset$. The unique enclave id is taken to be a software component of the Trusted Computing Base, generated during installation. The output of computations (through

Functionality $G_{\mathsf{att}}[\Sigma, \mathsf{reg}, \lambda]$

State variables	Description
vk	Master verification key, available to enclave programs
msk	Master secret key, protected by the hardware
$\mathcal{T} \leftarrow \emptyset$	Table for installed programs

```
On message INITIALIZE from a party P:
    let $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda), \mathsf{vk} \leftarrow \mathsf{spk}, \mathsf{msk} \leftarrow \mathsf{ssk}$
On message GETPK from a party P:
    return $\mathsf{vk}$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from a party $P$ where $P.\mathsf{pid} \in \mathsf{reg}$:
    if $P$ is honest then assert $\mathsf{idx} = P.\mathsf{sid}$
    generate nonce $\mathsf{eid} \xleftarrow{\$} \{0,1\}^\lambda$
    store $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}, \emptyset)$
    return $\mathsf{eid}$
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ from a party $P$ where $P.\mathsf{pid} \in \mathsf{reg}$:
    let $(\mathsf{idx}, \mathsf{prog}, \mathsf{mem}) \leftarrow \mathcal{T}[\mathsf{eid}, P]$, abort if not found
    let $(\mathsf{output}, \mathsf{mem}') \leftarrow \mathsf{prog}(\mathsf{input}, \mathsf{mem})$
    store $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}, \mathsf{mem}')$
    let $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{msk}, (\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output}))$ and return $(\mathsf{output}, \sigma)$
```

Figure 2.1: The G_{att} functionality of [230]

resume) consists of the (anonymous) ID of the enclave, the UC session ID, some unique encoding for the code computed by the enclave (which could be its source code, or its hash), and the output of the computation itself. Input does not have to be included in the attested return value, but if security requires parties to verify input, the function can return it as part of its output.

G_{att} is a Global Functionality in GUC [87] where the only meaningful global state shared between all protocols is the attestation verification key. This is a simplification over the EPID protocol that removes the key revocation phase. Attestation verification amounts to simply verifying the output data structure as described through a simple signature scheme with the globally available (both to machines with and without enclave capabilities) public verification key. The signing key is never released by the functionality, capturing the provisioning mechanism of the SGX system enclaves. The inclusion of the session ID in the attestation signature ensures that enclaves installed in different sessions (for which the simulator has no visibility) can not adversely interacts with the protocol.

As part of their work Pass, Shi, and Tramèr [230] show that TEE-assisted two-party computation is realisable in UC only if both parties have access to attested execution, and fair two-party computation is also possible if additionally both secure processors have access to a trusted clock.

Since its publication, numerous cryptographic protocols that rely on TEEs have been proven using G_{att} in the (G)UC framework [310, 98, 94, 309, 95, 301, 186, 182, 103, 148, 195, 185, 300, 158, 166, 134, 135, 224, 50, 101] or using [194] the Abstract Cryptography framework [206], and it as provides a basis for formalising TEEs in property-based definitions [210, 196, 124, 303, 130, 109, 269].

Additionally, some attempts have been made to relax the G_{att} functionality for the purposes of capturing TEE vulnerabilities. Tramer et al. [276] introduced the concept of transparent enclaves to model confidentiality leaks in an enclave program (formalised under GUC in [229, Section 8.1]). The transparent enclave functionality behaves exactly as G_{att}, except that for each RESUME operation, the functionality additionally leaks the randomness used by the enclave (allowing the OS to derive any secret created within the enclave). Since authenticity is still preserved, as the signing key for the enclave platform is not leaked, transparent enclaves are still useful for proving various constructions, such as zero-knowledge proofs and commitment schemes.

This is perhaps an excessively strong model, as the use of side channel attacks might only allow a portion of the memory or randomness to be learned by the adversary. Dörre, Mechler, and Müller-Quade [122] proposes both a weaker and a stronger variants. Since the SGX quoting enclave that allows producing attestation does not have any specific hardening mechanism compared to other enclaves running on the machine, besides being carefully implemented with side-effect free primitives, the authors argue that it is realistic to model a class of TEEs where side channels do not affect certain secure operations such as key exchange and symmetric encryption (since the quoting enclave relies on them for attestation to be successful). As such, they define *almost-transparent enclaves* as transparent enclaves with access to side-channel free implementations of symmetric cryptography primitives and Diffie-Hellman key exchange operation. On a resume operation, an almost transparent enclave leaks the random bits used during its execution, the memory of the enclave at the start of the resume call, and the return value of the cryptographic operations, but crucially not the randomness used to perform the cryptograhic functions. This allows the adversary (and the simulator) to learn any values that would have been leaked through any intermediate computation on secrets the enclave had access to. Additionally, they consider a *semi-honest enclave*, inspired by the modelling of [190], where the adversary is able to adaptively leak the list of operations executed by an enclave run by any party regardless of their corruption status. A semi-honest enclave model captures a scenario where the manufacturer of the TEE might have introduced a backdoor that enables them to remotely instruct any TEE-enabled machine to record and leak their data. Besides providing the alternative attacker models, their global functionalities are realised in UCGS, and allow any party to install an enclave (i.e. there is no fixed registry set reg).

The models of almost-transparent and semi-honest enclaves is motivated by the design of a protocol to implement one-sided Private Set Intersection (a two-party protocol where only one party learns the intersection of the two inputs). Dörre, Mechler, and Müller-Quade construct a protocol that realises one-sided PSI in the almost-transparent setting where one party is corrupted, and in the semi-honest when sitting where neither party is corrupted.

2.3 Cryptographic Primitives and functionalities

In this section, we formally define basic notation and definitions of cryptographic notions, and provide an overview of the primitives and ideal functionalities that are invoked by our constructions in later chapters. We give an overview for each functionality that should be sufficient for clarifying its communication interface with any interacting entities and resulting behaviour. For a more detailed description of the successive functionalities, we refer readers to the relevant work.

Notation. For a bit-string x, $|x|$ denotes the length of x, and λ denotes the security parameter. For a distribution D over a set $\mathcal{X}$, $x \leftarrow D$, denotes sampling an element $x \in \mathcal{X}$, according to D, and $x \leftarrow \mathcal{X}$ denotes sampling a uniform element x from $\mathcal{X}$. "$\approx$" denotes computational indistinguishability, and $\mathsf{negl}(\lambda)$ denotes an unspecified, negligible function, such that $\mathsf{negl}(\lambda) \leq \frac{1}{\lambda^c}$ for all $c \in \mathbb{R}$.

For an algorithm $\mathcal{A}$, using $y \leftarrow \mathcal{A}(x)$ we denote the execution of $\mathcal{A}$ on input x, receiving output y. In case $\mathcal{A}$ is randomized, y is a random variable and $\mathcal{A}(x;r)$ denotes the execution of $\mathcal{A}$ on input x with randomness r. We leave the randomness implicit for most randomised algorithms, except when relevant. An algorithm $\mathcal{A}$ is probabilistic polynomial-time (PPT) if $\mathcal{A}$ is randomized and for any $x, r \in \{0,1\}^*$, the computation of $\mathcal{A}(x;r)$ terminates in a number of steps polynomial in $(|x| + |r|)$. We denote by $\mathcal{A}^{\mathcal{X}}$ an algorithm that has blackbox access to the oracle $\mathcal{X}$ as part of its execution.

2.3.1 Public-key encryption

In the current section we define public-key encryption for chosen plaintext (CPA) and chosen ciphertext (CCA) attacks. Note that in the latter, the adversary is allowed to access the decryption oracle even after receiving the challenge ciphertext, usually referred to as CCA2 secure encryption.

The syntax of a public-key encryption scheme is defined below.

PKE syntax. A *public-key encryption* (PKE) scheme is a triple of algorithms $\mathsf{PKE} = (\mathsf{PGen}, \mathsf{Enc}, \mathsf{Dec})$ with the following syntax:

- (*Key generation*): PGen receives a security parameter λ, and outputs a fresh key pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PGen}(1^\lambda)$.
- (*Encryption*): Enc receives a public key pk and a message m and produces a ciphertext ct.
- (*Decryption*): Dec receives a secret key sk and a ciphertext ct and outputs a message m.

In the following sections the security parameter will be implicit when calling PGen. A PKE scheme must satisfy the following correctness property.

Correctness. For any message m,

$$\Pr[(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PGen}(1^\lambda); \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk},\mathrm{m}); \mathrm{m}' \leftarrow \mathsf{Dec}(\mathsf{sk},\mathsf{ct}) : \mathrm{m} = \mathrm{m}'] = 1.$$

CPA security game. For any PPT adversary $\mathcal{A}$, $b \in \{0,1\}$ and PKE scheme PKE, we consider the following CPA security game, denoted by $\mathsf{G}^{\mathsf{cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda, b)$:

- $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PGen}(1^\lambda)$.
- $\mathcal{A}$ receives pk and oracle access to $\mathcal{O}_{\mathsf{enc}}(\cdot) := \mathsf{Enc}(\mathsf{pk},\cdot)$.
- $\mathcal{A}$ outputs $(\mathrm{m}_0, \mathrm{m}_1)$.
- $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathrm{m}_b)$.
- $\mathcal{A}$ receives ct and oracle access to $\mathcal{O}_{\mathsf{enc}}(\cdot)$
- $\mathcal{A}$ outputs b'.
- Output $b' = b$.

Definition 2.2 (CPA security)**.** *A public-key encryption scheme* PKE *is* CPA-secure *if for all PPT adversaries* $\mathcal{A}$*, there exists a negligible function* negl *such that*

$$\left|\Pr[\mathsf{G}^{\mathsf{cpa}}_{\mathsf{PKE},\mathcal{A}}(\lambda,0) = 1] - \Pr[\mathsf{G}^{\mathsf{cpa}}_{\mathsf{PKE},\mathcal{A}}(\lambda,1)] = 1]\right| \leq \mathsf{negl}(\lambda).$$

CCA security game. For any PPT adversary $\mathcal{A}$, $b \in \{0,1\}$ and PKE scheme PKE, we consider the following CCA security game, denoted by $\mathsf{G}^{\mathsf{cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda, b)$:

- $(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{PGen}(1^\lambda)$.
- $\mathcal{A}$ receives pk and oracle access to $\mathcal{O}_{\mathsf{dec}}(\cdot) := \mathsf{Dec}(\mathsf{sk},\cdot)$.
- $\mathcal{A}$ outputs $(\mathrm{m}_0, \mathrm{m}_1)$.
- $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathrm{m}_b)$.
- $\mathcal{A}$ receives ct and oracle access to $\mathcal{O}_{\mathsf{dec}}(\cdot)$ but is not allowed to query ct.
- $\mathcal{A}$ outputs b'.

- Output $b' = b$.

Definition 2.3 (CCA security)**.** *A public-key encryption scheme* PKE *is* CCA-secure *if for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl *such that*

$$\left|\Pr[\mathsf{G}^{\mathsf{cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda,0) = 1] - \Pr[\mathsf{G}^{\mathsf{cca}}_{\mathsf{PKE},\mathcal{A}}(\lambda,1)] = 1]\right| \leq \mathsf{negl}(\lambda).$$

2.3.2 Digital Signatures

In the current section we define existential unforgeability against chosen message attacks (EU-CMA).

Digital signatures syntax. A *digital signature* (DS) scheme is a triple of algorithms $\Sigma = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ with the following syntax:

- (*Key generation*): Gen receives a string 1^λ for security parameter λ, and outputs a fresh public and secret keypair $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{Gen}(1^\lambda)$.
- (*Signing*): Sign receives a signing key ssk and a message m and produces a signature $\sigma \leftarrow \mathsf{Sign}(\mathsf{ssk}, m)$.
- (*Verification*): Vrfy receives a public key spk, a message m and a signature σ, and outputs a bit $b \leftarrow \mathsf{Vrfy}(\mathsf{spk}, m, \sigma)$.

In the following sections the security parameter will be implicit when calling Gen.

A DS scheme must satisfy the following correctness property.

Correctness. For any message m,

$$\Pr[(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{Gen}(1^\lambda); \sigma \leftarrow \mathsf{Sign}(\mathsf{ssk}, m) : \mathsf{Vrfy}(\mathsf{spk}, m, \sigma) = 1] \;=\; 1.$$

Unforgeability game. For any PPT adversary $\mathcal{A}$ and DS scheme Σ, we consider the following security game, denoted by $\mathsf{G}^{\mathsf{eu\text{-}cma}}_{\Sigma,\mathcal{A}}(\lambda)$:

- $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{Gen}(1^\lambda)$.
- $\mathcal{A}$ receives spk and oracle access to $O_{\mathsf{sign}}(\cdot) := \mathsf{Sign}(\mathsf{ssk}, \cdot)$. Let Q be the set of queries made by $\mathcal{A}$.
- $\mathcal{A}$ outputs (m, σ).

- If $\mathsf{Vrfy}(\mathsf{spk}, m, \sigma) = 1$ and $m \notin Q$, output 1, otherwise, output 0.

Definition 2.4 (Unforgeability). *A DS scheme Σ is* EU-CMA-secure*, if for all PPT adversaries $\mathcal{A}$, there exists a negligible function* negl *such that*

$$\Pr\left[\mathsf{G}^{\text{eu-cma}}_{\Sigma,\mathcal{A}}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda).$$

2.3.3 Non-Interactive Zero Knowledge (NIZK) proofs

Definition 2.5 (Robust NIZK [110]). *Let $\mathcal{W}$ be a witness relation for a language $\mathcal{L} \in \mathsf{NP}$. A non-interactive zero-knowledge argument system for $\mathcal{W}$ is a vector of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V}, \mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2))$, satisfying the following properties:*

- **Completeness**. *For any $x \in \mathcal{L}$ and any w such that $(x, w) \in \mathcal{W}$,*

$$\Pr\left[\mathcal{V}(\mathsf{crs}, x, \pi) = 1 \;\middle|\; \begin{array}{c} (\mathsf{crs}) \leftarrow \mathcal{G}(1^\lambda) \\ \pi \leftarrow \mathcal{P}(x, w, \mathsf{crs}) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda).$$

- **Zero-knowledge**. *For all non-uniform PPT adversaries $\mathcal{A}$, we have*

$$\Pr[\mathcal{A}^{\mathcal{P}(\mathsf{crs},\cdot,\cdot)}(\mathsf{crs}) = 1 \mid \mathsf{crs} \leftarrow \mathcal{G}(1^\lambda)] \approx \Pr[\mathcal{A}^{\mathcal{S}(\hat{\mathsf{crs}},\tau,\cdot,\cdot)}(\hat{\mathsf{crs}}) = 1 \mid (\hat{\mathsf{crs}}, \tau) \leftarrow \mathcal{S}_1(1^\lambda)],$$

  where $\mathcal{S}(\hat{\mathsf{crs}}, \tau, x, w) = \mathcal{S}_2(\hat{\mathsf{crs}}, \tau, x)$ if $(x, w) \in \mathcal{W}$, otherwise outputs $\perp$.

- **Simulation sound extractability**. *There exists a PPT algorithm $\mathcal{E}$, such that for all PPT algorithms $\mathcal{A}$, we have*

$$\Pr\left[\begin{array}{cc} \mathcal{V}(\hat{\mathsf{crs}}, x, \pi) = 1 & \wedge \\ w \notin \mathcal{W}(x) & \wedge \\ (x, \pi) \notin Q & \end{array} \;\middle|\; \begin{array}{c} (\hat{\mathsf{crs}}, \tau) \leftarrow \mathcal{S}_1(1^\lambda) \\ (x, \pi) \leftarrow \mathcal{A}^{\mathcal{S}_2(\hat{\mathsf{crs}},\tau,\cdot)}(\hat{\mathsf{crs}}) \\ w \leftarrow \mathcal{E}(\tau, x, \pi) \end{array}\right] \leq \mathsf{negl}(\lambda),$$

where Q denotes the simulation queries and answers (x_i, π_i), produced by the interaction between $\mathcal{A}$ and $\mathcal{S}_2$.

2.3.4 Repository Functionality

$\mathcal{REP}[W, \overline{R}]$ is based on the repository functionality of [203], parametrised by a writing party W and a set of reading parties $\overline{R}$

Functionality $\mathcal{REP}[W,\overline{R}]$

```
*On message* $(\textsc{write}, \mathrm{x})$ *from party* $W$*:*
    generate nonce $\mathsf{h} \xleftarrow{\$} \{0,1\}^\lambda$
    $M[\mathsf{h}] \leftarrow \mathrm{x}$
    **return** $\mathsf{h}$
*On message* $(\textsc{read}, \mathsf{h})$ *from party* $r \in \overline{R}$*:*
    **return** M[h]
```

2.3.5 Common Reference String Functionality

The $\mathcal{CRS}[D]$ functionality, based on the presentation of [46], is parametrised by a distribution D.

Functionality $\mathcal{CRS}[D]$

```
*On message* GET *from a party* $P$*:*
    **if** $\mathsf{crs} = \perp$ **then**
        **let** $\mathsf{crs} \leftarrow D$
    **return** $\mathsf{crs}$ to $P$
```

This functionality has a simple interface: on a request from any protocol party (or the adversary), a CRS string sampled from distribution D is returned. Once the CRS has been sampled, an instance of $\mathcal{CRS}^{\mathcal{G}}[D]$ will always return the same string. While this functionality is insufficient to realise global protocols in the GUC setting, where it is subsumed by the augmented CRS functionality [87], our usage of the functionality in the following sections is limited to local protocols. As a result, we are not concerned with the type of deniability attacks that are addressed by the new functionality.

2.3.6 Secure Channel Functionality

$\mathcal{SC}_R^S$ is the secure channel between source S and receiver R.

Functionality $\mathcal{SC}_R^S$

```
*On message* $(\textsc{send}, m)$ *from* $S$*:*
```

```
    let $M \leftarrow M \,||\, length(m)$
    return $(\text{SENT}, m)$ to $R$
On message GETMSGS from $\mathcal{A}$:
    output $M$ on the backdoor tape
```

This functionality provides some strong guarantees. The adversary is not activated upon sending, but can later on request the length of all messages sent through the channel.

Communication notation. By **send** (MSG, m) **to** $\mathcal{SC}_R^S$ and **receive** (MSG, m') , we denote the secure transmission of a message (MSG, m) from S to R. After sending the message S waits for the reply (MSG, m') over $\mathcal{SC}_S^R$. When the identity of the receiver or the sender is obvious from the context, we might use shorthands $\mathcal{SC}^S$ or $\mathcal{SC}_R$ respectively.

We write **send** (MSG, m) **to** $\mathcal{A}, p$ to denote a delayed (insecure) output to p. $\mathcal{A}$ is first informed about (MSG, m) and can then determine when and if to deliver the message to p.

2.3.7 The certification functionality $\mathcal{F}_{\mathsf{CERT}}$

We assume the existence of an ideal certification functionality $\mathcal{F}_{\mathsf{CERT}}$, inspired by the certification functionality and the certification authority functionality introduced in [77]. The difference between $\mathcal{F}_{\mathsf{CERT}}$ and the certification functionality in [77] is that (i) instead of taking over signature verification, $\mathcal{F}_{\mathsf{CERT}}$ allows the verifier to verify the validity of a signature offline, and (ii) it allows the generation of only one certificate (signature) for each party.

In particular, the functionality $\mathcal{F}_{\mathsf{CERT}}$ is parameterized by a set of parties $\mathcal{P}$ and an EUF-CMA signature scheme.

On a call to SIGN from a party in $\mathcal{P}$, the functionality signs the provided verification key if the party has not previously registered a key. On a call to GETK, it simply returns the verification key to allow the caller to verify the message offline.

Functionality $\mathcal{F}_{\mathsf{CERT}}[\mathcal{P}, \Sigma]$

State variables	Description
$\mathsf{spk}, \mathsf{ssk} \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$	Signature scheme parameters
$\mathcal{S} \leftarrow []$	List of registered party keys and corresponding certificates

```
On message GETK from a party P:
    return spk
On message (SIGN, vk) from P ∈ 𝒫:
    if $\mathcal{S}[P] = \bot$ then
        $\mathsf{cert} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, \mathsf{vk})$
        $\mathcal{S}[P] \leftarrow (\mathsf{cert}, \mathsf{vk})$
        return cert
```

2.3.8 Functional Encryption

Functional Encryption is a cryptographic primitive introduced by Boneh, Sahai, and Waters [59] as a generalisation over several pre-existing types of primitives, such as identity-based or attributed-based encryption, to compute some predicate over encrypted data. Since then, many variants have been proposed [199].

The standard notion of Functional Encryption consists of the following PPT algorithms over the class of functions $\mathtt{F} : \mathcal{X} \to \mathcal{Y}$:

- $\underline{\mathsf{KeyGen}}$: given security parameter 1^λ as input, KeyGen outputs master keypair $(\mathsf{mpk}, \mathsf{msk})$
- $\underline{\mathsf{Setup}}$: Setup takes $\mathsf{msk}, \mathrm{F} \in \mathtt{F}$ and returns functional key sk_F
- $\underline{\mathsf{Enc}}$: given string $\mathrm{x} \in \mathcal{X}$ and mpk, Enc returns ciphertext ct or an error
- $\underline{\mathsf{Dec}}$: on evaluation over some ciphertext ct and functional key sk_F, Dec returns $\mathrm{y} \in \mathcal{Y}$

Confidentiality A confidential Functional Encryption scheme allows only the function evaluation $\mathrm{F}(x)$ to be learned from the ciphertext ct and functional key sk_F.

Correctness A functional encryption scheme satisfies correctness if, for all functions $\mathrm{F} \in \mathtt{F}$ and all $\mathrm{x} \in \mathcal{X}$, the statement $\mathrm{F}(\mathrm{x}) \leftarrow \mathsf{Dec}(\mathsf{Enc}(\mathsf{mpk}, \mathrm{x}), \mathsf{sk}_\mathrm{F})$ holds.

Composable Functional Encryption Matt and Maurer [203] show that the notion of functional encryption is equivalent, up to assumed resources, to that of an access controlled repository, where some parties of type A are allowed to upload data, and other parties of type B are allowed to retrieve some function on that data, if they have received authorisation (granted by a party C). A party of type B does not learn anything else about the stored data, besides the function they are authorised to compute (and length leakage F_0). Composable functional encryption is impossible in the standard model, and their formulation require access to a random oracle. Badertscher et al. [35] defines the security property of consistency Functional Encryption in the composable setting, specifying different types of consistency based on which FE party is corrupted.

We now define the ideal functionality for functional encryption. Note that, our definition is along the lines of [35, 203], however, as opposed to [35], in which A and/or C might also get corrupted, we focus on the confidentiality of the encrypted message against a malicious decryptor, B. Yet, our techniques provide security against malicious encryptors, A,[7] satisfying the notion of *input consistency* from [35].

Thus for any function f and input x, our functionality guarantees that B learns only $f(x)$ (and any information that can be inferred from it).

Our treatment allows the existence of several parties of type B, A. When the functionality receives a message from such a party, their UC extended identity is used to distinguish who the sender is, and store or retrieve the appropriate data. For simplicity, in our ideal functionality we refer to all such parties as B, A, and the set of all such parties as $\mathbf{B}, \mathbf{A}$, respectively.

Functionality FE[$\mathtt{F}, \mathbf{A}, \mathbf{B}, \mathsf{C}$]

The functionality is parameterised by the function class $\mathtt{F} : \mathcal{X} \rightarrow \mathcal{Y}$, the set of extended identity for parties of type A and B, and the identity of authority party C.

State variables	Description
F_0	The distinguished leakage function
$M \leftarrow []$	Stores the plaintext and plaintext space for each message handler
$R \leftarrow []$	list of authorised functions for all t decryption parties

On message SETUP *from* C*:*

[7]Note that $\mathsf{B} \subseteq \mathsf{A}$.

```
    setup $\leftarrow \top$
    $R[i] \leftarrow \{F_0\}, \forall i \in \mathbf{B}$
    send SETUP to $\mathcal{A}$
On message $(\text{ENCRYPT}, x)$ from $P \in \mathbf{A} \cup \mathbf{B}$:
    if setup $= \top \wedge x \in \mathcal{X}$ then
        generate nonce h $\xleftarrow{\$} \{0,1\}^\lambda$
        $M[h] \leftarrow x$
        send $(\text{ENCRYPTED}, h)$ to $\mathcal{A}, P$
On message $(\text{KEYGEN}, F, B)$ from C:
    if $F \in$ F then
        $R[B] \leftarrow R[B] \cup \{F\}$
        send $(\text{ASSIGNED}, F, B)$ to $\mathcal{A}$, C
On message $(\text{DECRYPT}, h, F)$ from B $\in \mathbf{B}$:
    let $x \leftarrow M[h]$
    if A and C are honest then
        if $x \in \mathcal{X} \wedge F \in R[B]$ then
            return $(\text{DECRYPTED}, F(x))$
    else
        send $(\text{DECRYPT}, h, F, x)$ to $\mathcal{A}$ and receive $(\text{DECRYPTED}, y)$
        return $(\text{DECRYPTED}, y)$
```

Functional Encryption and TEEs In cryptography, hardware is frequently used to improve performance or circumvent impossibility results, e.g. [217, 10, 102]. As relevant examples, Desmedt and Quisquater [116] implements Identity-based encryption using tamperproof hardware, and Chung, Katz, and Zhou [100] show how to use stateless hardware tokens to implement functional encryption.

The IRON protocol by Fisch et al. [127] realises functional encryption by using an SGX-like TEE. Their protocol equips C with a key management enclave, and any number of B decryptors with a decryption and functional enclave. C generates a Public Key encryption keypair, and distributes the public key to A, who can use it to encrypt their data. To receive a functional key, B proves to C that it is running the correct enclaves through attestation, and its decryption enclave establishes a secure channel with the key management enclave to receive the secret key. We describe the protocol in more detail in Chapter 3. A further extension implementing verifiable functional encryption is presented in Suzuki et al. [268].

[44] proposes a new protocol to compute function-hiding FE. Their construction relies on the existance of a Multi-Input Functional Encryption scheme, whose decryption operation is executed obvliviously inside a TEE that has received an obfuscated functional key.

Chapter 3

Steel: Composable Hardware-based Stateful and Randomised Functional Encryption

Even people not working on TEEs are aware that SGX is broken. Actually, you don't even have to even know what SGX is to bring up the concern that their entire scheme relies on SGX working correctly.

Saagar Jha

This chapter introduces the new primitive of Functional Encryption for Stateful and Randomized functionalities (FESR), and constructs a new protocol to realise it. Our protocol, Steel is an extension of the Iron protocol of Fisch et al. [127], adapted to our new setting of stateful and randomised functionalities. We prove the security of Steel by showing it UC-realises the FESR ideal functionality, using the ideal functionality of Pass, Shi, and Tramèr [230] (PST) to capture the protocol's usage of TEEs. Our proof shows that it is possible to realise (in general) composable functional encryption as defined by Matt and Maurer [203] (see Section 2.3.8), while relying on TEEs instead of Random Oracles, and for the larger function class of FESR when additionally relying on a common reference string.

3.1 Technical Overview

Attested execution via the PST model. The Steel protocol assume access to the *global attestation functionality*, G_{att}, described in Section 2.2.4.1.

G_{att} is a UC functionality parameterised by a signature scheme, and a registry of all parties that are equipped with an attested execution processor. At a high level, G_{att} allows parties to register programs and ask for evaluations over arbitrary inputs, while also receiving signatures that ensure correctness of the computation. Since the manufacturer's signing key pair can be used in multiple protocols simultaneously, G_{att} is defined as a *global functionality* that uses the same key pair across sessions. In this chapter, we update the original GUC formulation of [230] to use the UCGS model (see Section 2.1.2.2)

Setting, adversarial model & security. Our treatment considers three types of parties that corresponds to the Functional Encryption roles described in Section 2.3.8: encryptors, denoted by A; decryptors, denoted by B; as well as a single party that corresponds to the trusted authority, denoted by C. The adversary is allowed to corrupt parties in B and request for evaluations of functions of its choice over messages encrypted by parties in A. We then require *correctness* of the computation, meaning that the state for each function has not been tampered with by the adversary, as well as *confidentiality* of the encrypted message, which ensures that the adversary learns only the output of the computation (and any information implied by it) and nothing more. Our treatment covers both stateful and randomized functionalities.

Steel**: UCGS-secure FE for stateful and randomized functionalities.** Steel is executed by the sets of parties discussed above, where besides encryptors, all other parties receive access to G_{att}, abstracting an execution in the presence of secure hardware enclaves. Our protocol is based on Iron [127], so we briefly revisit its main protocol operations: (1) **Setup**, executed by the trusted party C, installs a *key management enclave* (KME), running a program to generate *public-key encryption* and *digital signature*, key pairs. The public keys are published, while the equivalent secrets are kept encrypted in storage (using SGX's terminology, the memory is sealed). Each of the decryptors installs a *decryption enclave* (DE), and attests its authenticity to the KME to receive the secret key for the encryption scheme over a secure channel. (2) **KeyGen**, on input function F, calls KME, where the latter produces a signature on the measurement of an instantiated enclave that computes F, our functional key (an approach introduced by [92]). (3) When **Encrypt** is called by an encryptor, it uses the public encryption key to encrypt a message and sends the ciphertext to the intended recipients. (4) **Decrypt** is executed by a decrypting party seeking to compute some function F on a ciphertext. This operation instantiates a matching function enclave (or resume an existing one), whose role is that of computing the functional decryption, if an authorised functional key is provided.

Steel consists of the above operations, with the appropriate modifications to enable stateful functionalities. In addition, Steel provides some simplifications over the Iron protocol. In particular, we repurpose attestation's signature capabilities to supplant the need for a separate signature scheme to generate functional keys, and thus minimise the trusted computing base. In practice, a functional key for a function F can be produced by just letting the key generation process return F; as part of G_{att}'s execution, this produces an attestation signature σ over F, which becomes the functional key sk_{F} for that function, provided the generating enclave id is also made public (a requirement for verification, due to the signature syntax of attestation in G_{att}).

The statefulness of functional encryption is simply enabled by adding state storage to each functional enclave. Similar to [230], a curious artefact in the protocol's modeling is the addition of a "backdoor" that programs the output of the function evaluation subroutine, such that, if a specific argument is set on the input, the function evaluation returns the value of that argument. The reason for this addition is to enable simulation of signatures over function evaluations that have already been computed using the ideal functionality. We note that this addition does not impact correctness, as the state array is not modified if the backdoor is used, nor confidentiality, since the output of this subroutine is never passed to any other party besides the caller B. Finally, a further addition is that our protocol requires the addition of a proof of plaintext knowledge on top of the underlying encryption scheme. An efficient implementation for such a modification can be realised by drawing on the techniques of signed ElGamal [254], where the random coin used during encryption is sufficient to prove knowledge of the plaintext. However, this construction would require the use of the random oracle and generic group model, or at best the algebraic group model [132], making composition more difficult, and is thus not considered in full beyond this remark. The Steel protocol definition is presented in Section 3.3.

Security of Steel**.** Our protocol uses an existentially unforgeable under chosen message attacks (EU-CMA) signature scheme, Σ, a CCA-secure public-key encryption scheme, PKE, and a non-interactive zero knowledge scheme, N (as defined in Sections 2.3.2,2.3.2, and 2.3.3, respectively). Informally, Σ provides the guarantees required for realising attested computation (as discussed above), PKE is used to protect the communication between enclaves, and for protecting the encryptors' inputs. For the latter usage, it is possible to reduce the security requirement to CPA-security as we additionally compute a simulation-extractable NIZK proof of well-formedness of the ciphertext that guarantees non-malleability.

Our proof is via a sequence of hybrids in which we prove that the real world protocol execution w.r.t. Steel is indistinguishable from the ideal execution, in the presence of an ideal functionality that captures FE for stateful and randomized functionalities. The goal is to prove that the decryptor learns nothing more than an authorized function of the private input plaintext, thus our hybrids gradually fake all relevant information accessed by the adversary. In the first hybrid,[1] all signature verifications w.r.t. the attestation key are replaced by an idealized verification process, that only accepts message/signature pairs that have been computed honestly (i.e., we omit verification via Σ). Indistinguishability is proven via reduction to the EU-CMA security of Σ. Next we fake all ciphertexts exchanged between enclaves that carry the decryption key for the target ciphertext, over which the function is evaluated (those hybrids require reductions to the CCA security of PKE).[2] The next hybrid substitutes ZK proofs over the target plaintexts with simulated ones, and indistinguishability with the previous one reduces to the zero knowledge property of N. Then, for maliciously generated

[1]Here we omit some standard UC-related hybrids.

[2]Here CCA security is a requirement as the adversary is allowed to tamper with honestly generated ciphertexts.

ciphertexts under PKE – which might result via tampering with honestly generated encryptors' ciphertexts – instead of using the decryption operation of PKE, our simulator recovers the corresponding plaintext using the extractability property of N. Finally, we fake all ciphertexts of PKE, that encrypt the inputs to the functions (this reduces to CPA security). Note that, in [127], the adversary outputs the target message, which is then being encrypted and used as a parameter to the ideal world functionality that is accessed by the simulator in a black box way. In this work, we consider a stronger setting in which the adversary directly outputs ciphertexts of its choice. While in the classic setting for Functional Encryption (where Iron lives) simulation security is easily achieved by asking the adversarial enclave to produce an evaluation for the challenge ciphertext, in FESR the simulator is required to conduct all decryptions through the ideal functionality, so that the decryptor's state for that function can be updated. We address the above challenge by using the extractability property of NIZKs: for maliciously generated ciphertexts our simulator extracts the original plaintext and asks the ideal FESR functionality for its evaluation. Simulation-extractable NIZK can be efficiently instantiated, e.g., using zk-SNARKs [38]. Security of our protocol is formally proven in Section 3.4. The simulator therein provided could be easily adapted to show that the Iron protocol UCGS-realises Functional Encryption, by replacing the NIZK operations for maliciously generated ciphertexts with a decryption from the enclave, as described above.

3.2 Functional encryption for stateful and randomized functionalities

In this section we define the ideal functionality of functional encryption for *stateful* and *randomized functionalities* (FESR), a generalisation over Functional Encryption.

The syntax for this new primitive matches that of Functional Encryption schemes (outlined in Section 2.3.8). The two primitives differ by the parameterisation of the class of computable functions F; in the case of FESR, this is defined as

$$\mathtt{F} : \mathcal{X} \times \mathcal{S} \times \mathcal{R} \to \mathcal{Y} \times \mathcal{S}$$

where $\mathcal{S} = \{0,1\}^{s(\lambda)}, \mathcal{R} = \{0,1\}^{r(\lambda)}$ for polynomials $s(\cdot)$ and $r(\cdot)$.

The definition of the primitive follows from the ideal functionality, given below.

Our treatment considers the existence of several parties of type A (encryptors),

B (decryptors), and a singular trusted authority C. The latter is allowed to run the $\mathsf{KeyGen}, \mathsf{Setup}$ algorithms; parties of type A run Enc, and those of type B run Dec. The set of all decryptors (resp. encryptors) is denoted by $\mathbf{B}$ (resp. $\mathbf{A}$). When the functionality receives a message from such a party, their UC extended identity is used to distinguish who the sender is and store or retrieve the appropriate data. For simplicity, in our ideal functionality we refer to all parties by their type, with the implied assumption that it might refer to multiple distinct UC parties. For the sake of conciseness, we also omit including the sid parameter as an argument to every message.

The functionality reproduces the four algorithms that comprise functional encryption. During KeyGen, a record $\mathcal{P}$ is initialised for all t instances of B, to record the authorised functions for each instance, and its state. The Setup call marks a certain B as authorised to decrypt function F, and initialises its state to $\emptyset$. The Enc call allows a party A, B, to provide some input x, and receive a unique identifying handle h. This handle can then be provided, along with some F, to a decryption party to obtain an evaluation of F on the message stored therein. Performing the computation will also result in updating the state stored in $\mathcal{P}$.

Functionality FESR$[\mathtt{F}, \mathbf{A}, \mathbf{B}, \mathsf{C}]$

The functionality is parameterized by the randomized function class $\mathtt{F}$ such that for each $\mathrm{F} \in \mathtt{F} : \mathcal{X} \times \mathcal{S} \times \mathcal{R} \to \mathcal{Y} \times \mathcal{S}$, over state space $\mathcal{S}$ and randomness space $\mathcal{R}$, and by three distinct types of party identities $\mathsf{A} \in \mathbf{A}, \mathsf{B} \in \mathbf{B}, \mathsf{C}$ interacting with the functionality via dummy parties (that identify a particular role). For each decryptor/function pair, a state value is recorded.

State variables	Description
F_0	Leakage function returning the length of the message
$\mathsf{setup}[\cdot] \leftarrow \mathsf{false}$	Table recording which parties were initialized.
$\mathcal{M}[\cdot] \leftarrow \perp$	Table storing the plaintext for each message handler
$\mathcal{P}[\cdot] \leftarrow \perp$	Table of authorized functions and their states for all decryption parties

On message $(\textsc{setup}, P)$ *from party* C, *for* $P \in \mathbf{A} \cup \mathbf{B}$*:*

```
  $\mathsf{setup}[P] \leftarrow \mathsf{true}$
  **send** $(\textsc{setup}, P)$ **to** $\mathcal{A}$
```

On message $(\textsc{setup}, P)$ *from* $\mathcal{A}$, *for* $P \in \{\mathsf{A}, \mathsf{B}\}$*:*

```
  $\mathsf{setup}[P] \leftarrow \mathsf{true}$
```

```
    $\mathcal{P}[P, \mathrm{F}_0] \leftarrow \emptyset$
    **send** SETUP **to** $P$
  *On message* $(\text{ENCRYPT}, \mathrm{x})$ *from party* $P \in \{\mathsf{A}, \mathsf{B}\}$*:*
    **if** $\mathsf{setup}[P] = \mathsf{true} \wedge \mathrm{x} \in \mathcal{X}$ **then**
        generate nonce $\mathsf{h} \xleftarrow{\$} \{0,1\}^{\lambda}$
        $\mathcal{M}[\mathsf{h}] \leftarrow \mathrm{x}$
        **send** $(\text{ENCRYPTED}, \mathsf{h})$ **to** $P$
  *On message* $(\text{KEYGEN}, \mathrm{F}, \mathsf{B})$ *from party* $\mathsf{C}$*:*
    **if** $\mathrm{F} \in \mathtt{F} \wedge \mathsf{setup}[\mathsf{B}] = \mathsf{true}$ **then**
        **send** $(\text{KEYGEN}, \mathrm{F}, \mathsf{B})$ **to** $\mathcal{A}$ and **receive** ACK
        $\mathcal{P}[\mathsf{B}, \mathrm{F}] \leftarrow \emptyset$
        **send** $(\text{ASSIGNED}, \mathrm{F})$ **to** $\mathsf{B}$
  *On message* $(\text{DECRYPT}, \mathsf{h}, \mathrm{F})$ *from party* $\mathsf{B}$*:*
    $\mathrm{x} \leftarrow \mathcal{M}[\mathsf{h}]$
    **if** $\mathsf{C}$ is honest **then**
        **if** $\mathcal{P}[\mathsf{B}, \mathrm{F}] \neq \perp \wedge \mathrm{x} \in \mathcal{X}$ **then**
            $\mathrm{r} \leftarrow \mathcal{R}$
            $\mathsf{s} \leftarrow \mathcal{P}[\mathsf{B}, \mathrm{F}]$
            $(\mathrm{y}, \mathsf{s}') \leftarrow \mathrm{F}(\mathrm{x}, \mathrm{s}, \mathrm{r})$
            $\mathcal{P}[\mathsf{B}, \mathrm{F}] \leftarrow \mathsf{s}'$
            **return** $(\text{DECRYPTED}, \mathrm{y})$
    **else**
        **send** $(\text{DECRYPT}, \mathsf{h}, \mathrm{F}, \mathrm{x})$ **to** $\mathcal{A}$ and **receive** $(\text{DECRYPTED}, \mathrm{y})$
        **return** $(\text{DECRYPTED}, \mathrm{y})$
```

The functionality is defined for possible corruptions of parties in **B**, **A**. If C is corrupted, we can no longer guarantee the evaluation to be correct, since C might authorize the adversary to compute any function in $\mathtt{F}$. In this scenario, we allow the adversary to learn the original message value x and to provide an arbitrary evaluation y.

In this work we primarily focus on the security guarantees provided by FE, which is confidentiality of the encrypted message against malicious decryptors, B. Yet, it provides security against malicious encryptors, A, thus it satisfies *input consistency*, originally introduced by [35] (in which A and/or C might also get corrupted).

Our definition is along the lines of [35, 203]; in order to allow stateful and randomized functions, we extend the function class with support for private state and randomness as above. Whenever B accesses a function on the data from the repository, the repository draws fresh randomness, evaluates the function on the old state (for the current B and function), updates the state according to the function evaluation, and returns the result.

The property of confidentiality for functional encryption also holds for FESR, as the decrypting party is only allowed to learn the function evaluation (and not the state, before or after decryption). Correctness for FESR is slightly stronger than in Section 2.3.8: it is necessary for the state at any decryption to be uniquely determined by the sequence of previous decryption for the same F, B pair (without allowing B to influence its value, besides the choice of which ciphertexts to decrypt). Intuitively, the ideal world accessed controlled repository presented models both confidentiality and correctness. by inspection of the four lines $\mathsf{r} \leftarrow \mathcal{R}$, $\mathsf{s} \leftarrow \mathcal{P}[\mathsf{B},\mathsf{F}]$, $(\mathsf{y},\mathsf{s}') \leftarrow \mathsf{F}(\mathsf{x},\mathsf{s},\mathsf{r})$, and $\mathcal{P}[\mathsf{B},\mathsf{F}] \leftarrow \mathsf{s}'$.

In addition, our definition is the first one that captures stateful and randomized functionalities, where the latter refers to the standard notion of randomized functionalities in which each invocation of the function uses independent randomness. Therefore, our protocol achieves a stronger notion of randomized FE than [8, 146, 169], which require a new functional key for each invocation of the function, i.e., decryptions with the same functional key always return the same output. Our construction of functional keys through signatures over a description of the function body is facilitated by the hardware setup, as in [128]; this technique had previously been developed for FE schemes based on extractability [61] and indistinguishability [92] obfuscation.

3.3 A UC-formulation of Steel

In this section we present Steel in the UCGS setting. As we already state above, our treatment involves three roles: the *key generation* party C, the *decryption* parties $\mathbf{B}$, and the *encryption* parties $\mathbf{A}$. Among them, only the encryptor does not need to have access to an enclave. Confidentiality and correctness of the protocol in the face of an adversarial B hold from the proof of indistinguishability between real and ideal world in 3.4. We do not give any guarantees of security for corrupted A, C; although we remark informally that, as long as its enclave is secure, a corrupted C has little chances of learning the secret key. Besides the evaluation of any function in $\mathtt{F}$ it authorises itself to decrypt, it can also fake or extract from proofs of ciphertext validity π by authorizing a fake reference string crs. Before formally presenting our protocol we highlight important assumptions and conventions:

- For simplicity of presentation, we assume a single instance each for A, B (and therefore we use the ideal repository $\mathcal{REP}[\mathsf{A},\mathsf{B}]$, or $\mathcal{REP}$ for conciseness)
- all communication between parties (α,β) occurs over secure channels $\mathcal{SC}_{\alpha}^{\beta}, \mathcal{SC}_{\beta}^{\alpha}$. Our usage of this strong channel functionality is in line with the model of [203] (and [35])[3].
- Functional keys are (attestation) signatures by an enclave $\mathsf{prog}_{\mathsf{KME}}$ on input (keygen, F) for some function F; it is easy, given a list of keys, to retrieve the one which authorises decryptions of F
- keyword **fetch** retrieves a stored variable from memory and **aborts** if the value is not found
- on keyword **assert** the program checks that an expression is true, and proceeds to the next line, aborting otherwise
- all variables within an enclave are erased after use, unless saved to encrypted memory through the **store** keyword
- We use the notation "$\mathsf{output} \leftarrow G_{\mathsf{att}}.\mathsf{command}(\mathsf{input})$" as a shorthand for "**send** $(\mathsf{command}, \mathsf{input})$ **to** G_{att} and **receive** output "

Protocol Steel is parameterised by a function family $\mathtt{F} : \mathcal{X} \times \mathcal{S} \times \mathcal{R} \to \mathcal{Y} \times \mathcal{S}$, UC parties A, B, C, a CCA secure public key encryption scheme PKE, a EU-CMA secure signature scheme Σ, a Robust non-interactive zero-knowledge scheme N, and security parameter λ.

Protocol $\mathsf{Steel}[\mathtt{F},\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{PKE},\Sigma,\mathsf{N},\lambda]$

State variables	Description
$\mathsf{mpk} \leftarrow \perp$	Local copy of master public key for participants
$\mathsf{prog}_{\{\mathsf{KME},\mathsf{DE},\mathsf{FE}\}} \leftarrow \ldots$	Source code of enclaves as defined below
$\mathcal{K}[\cdot] \leftarrow \emptyset$	Table of function keys at B

Key Generation Authority C:

On message $(\textsc{Setup}, P)$:

[3]Like these works, we focus on the security guarantees against a corrupted party B, and this secure channel formulation simplifies the simulation of network interactions. We do, however, not see any fundamental obstacles to adopting a more realistic secure channel functionality such as that of [75]

```
  **if** $\mathsf{mpk} = \bot$ **then**
      $\mathsf{eid_{KME}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{C.sid}, \mathsf{prog_{KME}})$
      **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
      $(\mathsf{mpk}, \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{KME}}, (\text{init}, \mathsf{crs}, \mathsf{C.sid}))$
  **if** $P = \mathsf{A}$ **then**
      **send** $(\textsc{setup}, \mathsf{mpk})$ **to** $\mathcal{SC}_{\mathsf{A}}$
  **else if** $P = \mathsf{B}$ **then**
      **send** $(\textsc{setup}, \mathsf{mpk}, \mathsf{eid_{KME}})$ **to** $\mathcal{SC}_{\mathsf{B}}$ and **receive** $(\textsc{provision}, \sigma, \mathsf{eid_{DE}}, \mathsf{pk}_{KD})$
      $(\mathsf{ct_{key}}, \sigma_{\mathsf{sk}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{KME}}, (\text{provision}, (\sigma, \mathsf{eid_{DE}}, \mathsf{pk}_{KD}, \mathsf{eid_{KME}})))$
      **send** $(\textsc{provision}, \mathsf{ct_{key}}, \sigma_{\mathsf{sk}})$ **to** $\mathcal{SC}_{\mathsf{B}}$
*On message* $(\textsc{Keygen}, \mathrm{F}, \mathsf{B})$*:*
  **assert** $\mathrm{F} \in \mathtt{F} \wedge \mathsf{mpk} \neq \bot$
  $((\text{keygen}, \mathrm{F}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{KME}}, (\text{keygen}, \mathrm{F}))$
  $\mathsf{sk_F} \leftarrow \sigma$; **send** $(\textsc{keygen}, (\mathrm{F}, \mathsf{sk_F}))$ **to** $\mathcal{SC}_{\mathsf{B}}$
**Encryption Party** $\mathsf{A}$**:**
*On message* $(\textsc{Setup}, \mathsf{mpk})$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
  **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
  **store** $\mathsf{mpk}, \mathsf{crs}$; **return** $\textsc{setup}$
*On message* $(\textsc{encrypt}, \mathrm{m})$*:*
  **assert** $\mathsf{mpk} \neq \bot \wedge \mathrm{m} \in \mathcal{X}$
  $\mathsf{ct} \overset{\mathrm{r}}{\leftarrow} \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m})$
  $\pi \leftarrow \mathcal{P}((\mathsf{mpk}, \mathsf{ct}), (\mathrm{m}, \mathrm{r}), \mathsf{crs}), \mathsf{ct_{msg}} \leftarrow (\mathsf{ct}, \pi)$
  **send** $(\textsc{write}, \mathsf{ct_{msg}})$ **to** $\mathcal{REP}$ and **receive** $\mathsf{h}$
  **return** $(\textsc{encrypted}, \mathsf{h})$
**Decryption Party** $\mathsf{B}$**:**
*On message* $(\textsc{Setup}, \mathsf{mpk}, \mathsf{eid_{KME}})$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
  **store** $\mathsf{mpk}$; $\mathsf{eid_{DE}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog_{DE}})$
  **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
  $((\mathsf{pk}_{KD}, \cdot, \cdot), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{DE}}, (\text{init-setup}, \mathsf{eid_{KME}}, \mathsf{crs}, \mathsf{B.sid}))$
  **send** $(\textsc{provision}, \sigma, \mathsf{eid_{DE}}, \mathsf{pk}_{KD})$ **to** $\mathcal{SC}_{\mathsf{C}}$ and **receive** $(\textsc{provision}, \mathsf{ct_{key}}, \sigma_{\mathsf{KME}})$
  $G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{DE}}, (\text{complete-setup}, \mathsf{ct_{key}}, \sigma_{\mathsf{KME}}))$
  **return** $\textsc{setup}$
*On message* $(\textsc{keygen}, (\mathrm{F}, \mathsf{sk_F}))$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
  $\mathsf{eid_F} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog_{FE}}[\mathrm{F}])$
  $(\mathsf{pk_{DF}}, \sigma_{\mathsf{F}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_F}, (\text{init}, \mathsf{mpk}, \mathsf{B.sid}))$
```

```
    $\mathcal{K}[\mathrm{F}] \leftarrow (\sigma_{\mathsf{F}}, \mathsf{eid}_{\mathrm{F}}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}})$
    **return** $(\textsc{assigned}, F)$
*On message* $(\textsc{decrypt}, \mathrm{F}, \mathsf{h})$*:*
    **assert** $\mathcal{K}[\mathrm{F}] \neq \perp$
    **send** $(\textsc{read}, \mathsf{h})$ **to** $\mathcal{REP}$ and **receive** $\mathsf{ct}_{\mathsf{msg}}$
    $(\sigma_{\mathsf{F}}, \mathsf{eid}_{\mathsf{F}}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}}) \leftarrow \mathcal{K}[\mathrm{F}]$
    $((\mathsf{ct}_{\mathsf{key}}, \mathsf{crs}), \sigma_{\mathsf{DE}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\mathrm{provision}, \sigma_{\mathsf{F}}, \mathsf{eid}_{\mathsf{F}}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}}, \mathrm{F}))$
    $((\mathrm{computed}, \mathrm{y}), \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \perp))$
    **return** $(\textsc{Decrypted}, \mathrm{y})$

     $\underline{\mathsf{prog}_{\mathsf{KME}}}$:
     on input $(\mathrm{init}, \mathsf{crs}, \mathsf{idx})$:
         **assert** $\mathsf{pk} = \perp; (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.PGen}(1^{\lambda})$
         **store** $\mathsf{sk}, \mathsf{crs}, \mathsf{idx}$; **return** $\mathsf{pk}$
     on input $(\mathrm{provision}, (\sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}))$:
         $\mathsf{vk}_{\mathsf{att}} \leftarrow G_{\mathsf{att}}.\mathsf{vk}$; **fetch** $\mathsf{crs}, \mathsf{idx}, \mathsf{sk}$
         **assert** $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, (\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, (\mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}), \sigma_{\mathsf{DE}})$
         $\mathsf{ct}_{\mathsf{key}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{KD}, \mathsf{sk})$
         **return** $\mathsf{ct}_{\mathsf{key}}$
     on input $(\mathrm{keygen}, \mathrm{F})$:
         **return** $(\mathrm{keygen}, \mathrm{F})$
```

```
$\underline{\mathsf{prog}_{\mathsf{DE}}}$:
on input (init-setup, $\mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}, \mathsf{idx}$):
    assert $\mathsf{pk}_{KD} \neq \perp$
    $(\mathsf{pk}_{KD}, \mathsf{sk}_{KD}) \leftarrow \mathsf{PKE.PGen}(1^{\lambda})$
    store $\mathsf{sk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}, \mathsf{idx}$
    return $\mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}$
on input (complete-setup, $\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}$):
    $\mathsf{vk}_{\mathsf{att}} \leftarrow G_{\mathsf{att}}.\mathsf{vk}$
    fetch $\mathsf{eid}_{\mathsf{KME}}, \mathsf{sk}_{KD}, \mathsf{idx}$
    $\mathrm{m} \leftarrow (\mathsf{idx}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, \mathsf{ct}_{\mathsf{key}})$
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \mathrm{m}, \sigma_{\mathsf{KME}})$
    $\mathsf{sk} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{KD}, \mathsf{ct}_{\mathsf{key}})$
    store $\mathsf{sk}, \mathsf{vk}_{\mathsf{att}}$
on input (provision, $\sigma, \mathsf{eid}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}}, \mathrm{F}$):
    fetch $\mathsf{eid}_{\mathsf{KME}}, \mathsf{vk}_{\mathsf{att}}, \mathsf{sk}, \mathsf{idx}, \mathsf{crs}$
    $\mathrm{m}_1 \leftarrow (\mathsf{idx}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, (\mathrm{keygen}, \mathrm{F}))$
    $\mathrm{m}_2 \leftarrow (\mathsf{idx}, \mathsf{eid}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}], \mathsf{pk}_{\mathsf{DF}})$
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \mathrm{m}_1, \mathsf{sk}_{\mathrm{F}})$ and
    $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \mathrm{m}_2, \sigma)$
    return $\mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}), \mathsf{crs}$
```

```
$\underline{\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]}$:
on input (init, $\mathsf{mpk}, \mathsf{idx}$):
    assert $\mathsf{pk}_{\mathsf{DF}} = \perp$
    $(\mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathsf{DF}}) = \mathsf{PKE.PGen}(1^{\lambda})$
    $\mathsf{mem} \leftarrow \emptyset$; store $\mathsf{sk}_{\mathsf{DF}}, \mathsf{mem}, \mathsf{mpk}, \mathsf{idx}$
    return $\mathsf{pk}_{\mathsf{DF}}$
on input (run, $\sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \mathsf{y}'$):
    if $\mathsf{y}' \neq \perp$
        return (computed, $\mathsf{y}'$)
    $\mathsf{vk}_{\mathsf{att}} \leftarrow G_{\mathsf{att}}.\mathsf{vk}; (\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
    fetch $\mathsf{sk}_{\mathsf{DF}}, \mathsf{mem}, \mathsf{mpk}, \mathsf{idx}$
    $\mathrm{m} \leftarrow (\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{crs})$
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \mathrm{m}, \sigma_{\mathsf{DE}})$
    $\mathsf{sk} = \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{DF}}, \mathsf{ct}_{\mathsf{key}})$
    assert $\mathsf{N}.\mathcal{V}((\mathsf{mpk}, \mathsf{ct}), \pi, \mathsf{crs})$
    $\mathrm{x} = \mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$
    $\mathsf{out}, \mathsf{mem}' = \mathrm{F}(\mathrm{x}, \mathsf{mem})$
    store $\mathsf{mem} \leftarrow \mathsf{mem}'$
    return (computed, out)
```

As we mention in the Introduction, our modeling considers a “backdoor” in the $\mathsf{prog}_{\mathsf{FE}}$.run subroutine, such that, if the last argument is set, the subroutine just returns the value of that argument, along with a label declaring the function was evaluated. The addition of the label “computed” is necessary, otherwise the backdoor would allow producing an attested value for the public key generated in subroutine $\mathsf{prog}_{\mathsf{FE}}$.init.

As a further addition we strengthen the encryption scheme with a plaintext proof of knowledge (PPoK). For public key pk, ciphertext ct, plaintext m, ciphertext randomness r, the relation $R = \{(\mathsf{pk}, \mathsf{ct}), (\mathrm{m}, \mathrm{r}) | \mathsf{ct} = \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m}; \mathrm{r})\}$ defines the language L_R of correctly computed ciphertexts. As a CPA secure PKE scheme becomes CCA secure when extended with a simulation-extractable PPoK[187], this is a natural equivalence to the CCA security requirement of Iron. Additionally, it enables the simulator to extract valid plaintexts from all adversarial ciphertexts. In our security proof the simulator will submit these plaintexts to FESR on behalf of the corrupt B to keep the decryption states of the real and ideal world synchronized.

3.4 UC-security of Steel

We now prove the security of Steel in the UCGS framework. To make the PST model compatible with the UCGS model, we first define the identity bound ξ.

The identity bound ξ on the environment. Our restrictions are similar to those in PST [229, Section 3.2], namely we assume that the environment can access G_{att} in the following ways: (1) Acting as a corrupted party, and (2) acting as an honest party but only for non-challenge protocol instances.

We now prove our main theorem.

Theorem 3.1. Steel *UC-realises the* FESR *functionality in the presence of the global functionality* G_{att} *and local functionalities* $\mathcal{CRS}, \mathcal{REP}, \mathcal{SC}$*, with respect to the identity bound* ξ *defined above.*

We present a simulator algorithm $\mathcal{S}_{\text{FESR}}$ such that Theorem 2.1 holds for protocols Steel and $\mathsf{IDEAL}_{\text{FESR}}$ (the protocol encapsulating the ideal functionality and a set of dummy parties corresponding to the real-world parties in Steel). Following [203], our proof considers static corruption of a single party B, we did, however, not encounter any road-blocks to adaptive corruption of multiple decryptors besides increased proof notational complexity. The environment is unable to distinguish between an execution of the Steel protocol in the real world, and the protocol consisting of $\mathcal{S}_{\text{FESR}}$, dummy parties $\mathsf{A}, \mathsf{B}, \mathsf{C}$ and ideal functionality FESR. Both protocols have access to the shared global subroutines of G_{att}. While hybrid functionalities $\mathcal{REP}$, $\mathcal{SC}$, $\mathcal{CRS}$ (defined in Section 2.3.4,2.3.6, and 2.3.5 respectively) are only available in the real world and need to be reproduced by the simulator, we use $\mathcal{SC}$ to denote simulated channels.

Given protocols Steel, FESR, and G_{att}, Steel ξ-UC emulates FESR in the presence of G_{att} if $M[\mathsf{Steel}, G_{\mathsf{att}}]$ ξ-UC emulates $M[\text{FESR}, G_{\mathsf{att}}]$ (see Definition 2.1). We focus or exposition on the messages exchanged between the environment and the machine instances executing Steel, FESR, and G_{att}, since the machine M is simply routing messages from any external party to the right ITI

The simulator operates in the ideal world, where we have the environment $\mathcal{Z}$ sending message to dummy protocol parties which forward their inputs to the ideal functionality FESR. $\mathcal{S}_{\text{FESR}}$ is activated either by an incoming message from a corrupted party or the adversary, or when FESR sends a message to the ideal world adversary. As $\mathcal{A}$ is a dummy adversary which normally forwards all queries between the corrupt party and the environment, $\mathcal{S}_{\text{FESR}}$ gets to see all messages $\mathcal{Z}$ sends to $\mathcal{A}$. The simulator is allowed to send messages to the FESR and G_{att} functionalities impersonating corrupt parties. In the current setting, the only party that can be corrupted such that FESR still gives non trivial guarantees is party B. The notation $\mathsf{output} \leftarrow G_{\mathsf{att}}.\mathsf{command}(\mathsf{input})$ here is a shorthand for "**simulate sending** $(\mathsf{command}, \mathsf{input})$ **to** G_{att} **through** B and **receive** output". Thus, whenever the real world adversary or the ideal world simulator call $G_{\mathsf{att}}.\mathsf{install}$ and $G_{\mathsf{att}}.\mathsf{resume}$ for the challenge protocol instance, they must do so using the identity of B.

Simulator $\mathcal{S}_{\text{FESR}}[\mathsf{PKE}, \Sigma, \mathsf{N}, \lambda, \mathrm{F}]$

State variables	Description
$\mathcal{H}[\cdot] \leftarrow \emptyset$	Table of ciphertext and handles in public repository
$\mathcal{K} \leftarrow []$	List of $\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]$ enclaves and their $\mathsf{eid}_{\mathsf{F}}$
$\mathcal{G} \leftarrow \{\}$	Collects all messages sent to G_{att} and its response
$\mathcal{B} \leftarrow \{\}$	Collects all messages signed by G_{att}
$(\hat{\mathsf{crs}}, \tau) \leftarrow \mathsf{N}.\mathcal{S}_1$	Simulated reference string and trapdoor

```
For Key Generation Authority C:
On message (SETUP, P) from FESR:
  if $\mathsf{mpk} = \perp$ then
    $\mathsf{eid}_{\mathsf{KME}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{C.sid}, \mathsf{prog}_{\mathsf{KME}})$
    $(\mathsf{mpk}, \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, \mathrm{init}, \hat{\mathsf{crs}}, \mathsf{C.sid})$
  if $P = \mathsf{A}$ then
    send $(\textsc{setup}, \mathsf{mpk})$ to $\mathcal{SC}_{\mathsf{A}}$
  else if $P = \mathsf{B}$ then
    send $(\textsc{setup}, \mathsf{mpk}, \mathsf{eid}_{\mathsf{KME}})$ to $\mathcal{SC}_{\mathsf{B}}^{\mathsf{C}}$ and receive $(\textsc{provision}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$
    assert $(\mathsf{C.sid}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{pk}_{KD}) \in \mathcal{B}[\sigma]$
    $(\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathrm{provision}, (\sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \hat{\mathsf{crs}}))))$
    send $(\textsc{provision}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}})$ to $\mathcal{SC}_{\mathsf{B}}$

On message (KEYGEN, F, B) from FESR:
  assert $\mathrm{F} \in \mathrm{F} \wedge \mathsf{mpk} \neq \perp$
  $((\mathrm{keygen}, \mathrm{F}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathrm{keygen}, \mathrm{F}))$
  $\mathsf{sk}_{\mathrm{F}} \leftarrow \sigma$; $\mathcal{B}[\mathsf{sk}_{\mathrm{F}}] \leftarrow (\mathsf{C.sid}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, (\mathrm{keygen}, \mathrm{F}))$
  send $(\textsc{keygen}, (\mathrm{F}, \mathsf{sk}_{\mathrm{F}}))$ to $\mathcal{SC}_{\mathsf{B}}$

For Decryption Party B:
On message GET from party B to $\mathcal{CRS}$:
  **send** $(\mathrm{CRS}, \hat{\mathsf{crs}})$ **to** B
*On message* $(\textsc{read}, \mathsf{h})$ *from party* B *to* $\mathcal{REP}$:
  **send** $(\textsc{decrypt}, \mathrm{F}_0, \mathsf{h})$ **to** FESR on behalf of B and **receive** $|\mathrm{m}|$
  **assert** $|\mathrm{m}| \neq \perp$
  $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{mpk}, 0^{|\mathrm{m}|})$
  $\pi \leftarrow \mathsf{N}.\mathcal{S}_2(\hat{\mathsf{crs}}, \tau, (\mathsf{mpk}, \mathsf{ct}))$
  $\mathsf{ct}_{\mathsf{msg}} \leftarrow (\mathsf{ct}, \pi)$; $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] \leftarrow h$
  **send** $(\textsc{read}, \mathsf{ct}_{\mathsf{msg}})$ **to** B
*On message* $(\textsc{install}, \mathsf{idx}, \mathsf{prog})$ *from party* B *to* $G_{\mathsf{att}}$:
  $\mathsf{eid} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{idx}, \mathsf{prog})$
  $\mathcal{G}[\mathsf{eid}].\mathsf{install} \leftarrow (\mathsf{idx}, \mathsf{prog})$
  // $\mathcal{G}[\mathsf{eid}].install[1]$ is the program's code
  **forward** eid to B
*On message* $(\textsc{resume}, \mathsf{eid}, \mathsf{input})$ *from party* B *to* $G_{\mathsf{att}}$:
  // The $G_{\mathsf{att}}$ registry does not allow $B$ to access $\mathsf{eid}_{\mathsf{KME}}$ in real world
  **assert** $\mathcal{G}[\mathsf{eid}] \neq \perp \wedge \mathsf{eid} \neq \mathsf{eid}_{\mathsf{KME}}$
  **if** $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] \neq \mathsf{prog}_{\mathsf{FE}}[\cdot] \vee (\mathsf{input}[0] = \mathrm{run} \wedge \mathsf{input}[-1] \neq \perp)$ **then**
    $(\mathsf{output}, \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}, \mathsf{input})$
    $\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \,\|\, (\sigma, \mathsf{input}, \mathsf{output})$
    $\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], \mathsf{output})$
    **if** $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] = \mathsf{prog}_{\mathsf{DE}} \wedge \mathsf{input}[0] = \mathrm{provision}$ **then**
      $(\mathrm{provision}, \sigma_{\mathsf{DE}}, \mathsf{eid}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}}, \mathrm{F}) \leftarrow \mathsf{input}$
      **fetch** $(\cdot, (\mathrm{init\text{-}setup}, \mathsf{eid}_{\mathsf{KME}}, \hat{\mathsf{crs}}), \cdot) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
      **assert** $(\mathsf{idx}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, (\mathrm{keygen}, \mathrm{F})) \in \mathcal{B}[\mathsf{sk}_{\mathrm{F}}]$
      **assert** $(\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \hat{\mathsf{crs}}) \in \mathcal{B}[\sigma_{\mathsf{DE}}]$
    **forward** $(\mathsf{output}, \sigma)$ to B
  **else**
    $\mathsf{idx}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}] \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{install}$
    $(\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \hat{\mathsf{crs}}, \perp) \leftarrow \mathsf{input}$
    **assert** $(\sigma_{\mathrm{F}}, (\mathrm{init}, \mathsf{mpk}, \mathsf{idx}), (\mathsf{pk}_{\mathsf{DF}})) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
    **assert** $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}], \mathsf{pk}_{\mathsf{DF}}) \in \mathcal{B}[\sigma_{\mathrm{F}}]$
    **assert** $(\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \hat{\mathsf{crs}})) \in \mathcal{B}[\sigma_{\mathsf{DE}}]$
    // If the ciphertext was not computed honestly and saved to $\mathcal{H}$
    **if** $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] = \perp$ **then**
      $(\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
      $(\mathrm{m}, \mathrm{r}) \leftarrow \mathsf{N}.\mathcal{E}(\tau, (\mathsf{mpk}, \mathsf{ct}), \pi)$
```

```
        if $m = \perp$ then send $(\textsc{decrypt}, F, \perp)$ to $B$ and abort
        send $(\textsc{encrypt}, m)$ to FESR on behalf of $B$ and receive $h$
        $\mathcal{H}[\mathsf{ct_{msg}}] \leftarrow h$
    $h \leftarrow \mathcal{H}[\mathsf{ct_{msg}}]$
    send $(\textsc{decrypt}, F, h)$ to FESR on behalf of $B$ and receive $y$
    $((\mathrm{computed}, y), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_F, (\mathrm{run}, \perp, \perp, \perp, \perp, \perp, y))$
    $\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \,||\, (\sigma, \mathsf{input}, (\mathrm{computed}, y)))$
    $\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], (\mathrm{computed}, y))$
    forward $((\mathrm{computed}, y), \sigma)$ to $B$
```

3.4.1 Designing the simulation

The ideal functionality FESR and protocol Steel share the same interface consisting of messages SETUP, KEYGEN, ENCRYPT, DECRYPT. During Steel's SETUP, the protocol generates public parameters when first run, and provisions the encrypted secret key to the enclaves of B. As neither of these operations are executed by the ideal functionality, we need to simulate them, generating and distributing keys outside of party C.

As in Steel, we distribute the public encryption key on behalf of C to any newly registered B and A over secure channels. Once B has received this message, it will try to obtain the (encrypted) decryption key for the global PKE scheme from party C and its provision subroutine of $\mathsf{prog}_{\mathsf{KME}}$. Since C is a dummy party in the ideal world, it would not respond to this request, so we let $\mathcal{S}_{\mathrm{FESR}}$ respond. In Steel key parameters are generated within the key management enclave, and communication of the encrypted secret key to the decryption enclave produces an attestation signature. Thus, the simulator, which can access G_{att} impersonating B, is required to install an enclave. Because of the property of anonymous attestation, the environment cannot distinguish whether the new enclave was installed on B or C. If the environment tries to resume the program running under $\mathsf{eid}_{\mathsf{KME}}$ in the test session through B, this is intercepted and dropped by the simulator.

Before sending the encrypted secret key, the simulator verifies that B's public key was correctly produced by an attested decryption enclave, and was initialised with the correct parameters. If an honest enclave has been instantiated and we can verify that it uses $\mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}$, we can safely send the encrypted sk to the corrupted party as no one can retrieve the decryption key from outside the enclave.

On message $(\textsc{keygen}, F, B)$ from the functionality after a call to KEYGEN, $\mathcal{S}_{\mathrm{FESR}}$ simply produces a functional key by running the appropriate $\mathsf{prog}_{\mathsf{KME}}$ procedure through

G_{att}. Similarly, on receiving $(\textsc{read}, \mathsf{h})$ for $\mathcal{REP}$, $\mathcal{S}_{\text{FESR}}$ produces an encryption of a canonical message (a string of zeros) and simulates the response.

When the request to compute the functional decryption of the corresponding ciphertext is sent to $\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]$, we verify that the party B has adhered to the Steel protocol execution, aborting if any of the required enclave installation or execution steps have been omitted, or if any of the requests were made with dishonest parameters generated outside the enclave execution (we can verify this through the attestation of enclave execution). If the ciphertext was not obtained through a request to $\mathcal{REP}$, we use the NIZK extractor to learn the plaintext m and submit a message $(\textsc{encrypt}, \mathrm{m})$ to FESR on behalf of the corrupt B. This guarantees that the state of FESR is in sync with the state of $\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]$ in the real world.

If all such checks succeed, and the provided functional key is valid, $\mathcal{S}_{\text{FESR}}$ fetches the decryption from the ideal functionality. While the Steel protocol ignores the value of the attested execution of run, we can expect the adversary to check its result for authenticity. Therefore, it is necessary to pass the result of our decryption y through the backdoor we constructed in $\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]$. This will produce an authentic attestation signature on y, which will pass any verification check convincingly (as discussed in the previous section, the backdoor does not otherwise impact the security of the protocol).

3.4.2 Proof of Security

We proceed to show that the real world execution with respect to the Steel protocol and dummy adversary is indistinguishable from the ideal execution w.r.t. the ideal functionality FESR and the simulator described above. Let Hybrid 0 be the real world execution w.r.t. to the Steel protocol and dummy adversary.

Hybrid 1 We define Hybrid 1 to be identical to 0, but as translated to the ideal world; that is, we replace protocol Steel with a protocol consisting of dummy parties $\mathsf{A}, \mathsf{B}, \mathsf{C}$ and ideal functionality FESR; the dummy adversary machine is replaced by a simulator $\mathcal{S}'_{\text{FESR}}$, who emulates Steel's execution by emulating all operations that would normally be run by the honest parties. Messages are now sent to the ideal functionality, but the responses are ignored and the original protocol is perfectly executed by $\mathcal{S}'_{\text{FESR}}$. Hybrid 0 and 1 are indistinguishable because their behaviour is equivalent due to simulator $\mathcal{S}'_{\text{FESR}}$. Since the ideal functionality is not visible to the environment, it is harmless for the dummy parties to send messages to it.

Lemma 3.1. H_0 *is identical to* H_1.

Hybrid 2 diverges from Hybrid 1 in that all signature verifications obtained with the attestation public key $\mathsf{vk}_{\mathsf{att}}$ are replaced by the process of storing all outgoing messages from the G_{att} functionality in a map data structure, and checking on verification that the message and corresponding signature were correctly recorded. The behaviour of the two hybrids is equivalent, as long as the adversary in Hybrid 2 is not able to provide a signature such that the verification checks are successful, even if the messages were not recorded as coming through the G_{att} functionality. Assuming the unforgeability property of G_{att}'s signature scheme is satisfied, Hybrid 1 is then indistinguishable to Hybrid 2.

Lemma 3.2. *If* Σ *is EU-CMA secure then* $\mathsf{H}_1 \approx \mathsf{H}_2$*, over the randomness used by all parties in* H_1, H_2.

Proof. To prove the indistinguishability of the two hybrids, we show how an environment $\mathcal{Z}$ that breaks attestation can be used to build an adversary $\mathbb{R}$ that breaks unforgeability of the signature scheme.

For attestation to break, the adversary needs to produce signature σ such that $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \sigma, (\mathsf{sid}, \mathsf{eid}_{\mathsf{X}}, \mathsf{prog}_{\mathsf{X}}, \mathsf{out})) = 1$ for some program $\mathsf{prog}_{\mathsf{X}}$ and output out such that no execution of $\mathsf{prog}_{\mathsf{X}}$ under sid produced out with attestation σ

We now describe adversary $\mathbb{R}$, whose goal is to break signature game (defined in Section 2.3.2) $\mathcal{C}^{\Sigma}$ by submitting a forged signature.

Reduction $\mathbb{R}^{\mathcal{C}^{\Sigma}, \mathcal{Z}}$

State variables	Description
spk	Public key for game $\mathcal{C}^{\Sigma}$
$\mathcal{H} \leftarrow \{\}$	Repository of message requests
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$
$\mathsf{state} \leftarrow \{\}$	Dictionary to hold the state for each function

```
On message (SETUP, $P$):
    if $\mathsf{mpk} = \perp$ then
        $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{PKE.PGen}(1^{\lambda})$
        send GET to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
    if $P = \mathsf{A}$ then
```

```
        send $(\text{SETUP}, \mathsf{mpk})$ to $\mathcal{SC}_\mathsf{A}$
    else if $P = \mathsf{B}$ then
        send $(\text{SETUP}, \mathsf{mpk}, \mathsf{eid}_\mathsf{KME})$ to $\mathcal{SC}_\mathsf{B}$ and receive $(\text{PROVISION}, \sigma, \mathsf{eid}_\mathsf{DE}, \mathsf{pk}_{KD})$
        $\mathrm{m} \leftarrow (\mathsf{sid}, \mathsf{eid}_\mathsf{DE}, \mathsf{prog}_\mathsf{DE}, \mathsf{pk})$
        if $\Sigma.\mathsf{Vrfy}(\mathsf{spk}, \mathrm{m}, \sigma) \wedge \sigma \notin \mathcal{C}^\Sigma.\mathcal{Q}$ then
            output $(\mathrm{m}, \sigma)$ to $\mathcal{C}^\Sigma$
        $\mathsf{c} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}, \mathsf{msk})$
        $\sigma_\mathsf{KME} \leftarrow \mathcal{C}^\Sigma.\mathcal{O}_\mathsf{sign}(\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_\mathsf{KME}, \mathsf{c})$
        send $(\text{PROVISION}, c, \sigma_\mathsf{KME}))$ to $\mathcal{SC}_\mathsf{B}$
On message $(\text{KEYGEN}, \mathrm{F}, \mathsf{B})$:
    if $\mathrm{F} \notin \mathrm{F} \vee \mathsf{mpk} = \perp$ then return $\perp$
    $\sigma \leftarrow \mathcal{C}^\Sigma.\mathcal{O}_\mathsf{sign}(\mathrm{keygen}, \mathrm{F})$
    $\mathsf{state} \leftarrow \vec{0}$
    send $(\text{KEYGEN}, (\mathrm{F}, \sigma))$ to $\mathcal{SC}_\mathsf{B}$
On message $(\text{ENCRYPT}, m)$:
    if $\mathsf{mpk} = \perp \vee \mathrm{m} \notin \mathcal{X}$ then return $\perp$
    generate nonce $\mathsf{h} \xleftarrow{\$} \{0,1\}^\lambda; \mathcal{H}[\mathsf{h}] \leftarrow \mathrm{m}$
    return $(\text{ENCRYPTED}, \mathsf{h})$
On message $(\text{READ}, \mathsf{h})$ from party $\mathsf{B}$ to $\mathcal{REP}$:
    $\mathrm{m} \leftarrow \mathcal{H}[\mathsf{h}]$
    $\mathrm{r} \leftarrow \{0,1\}^\lambda; \mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m}; \mathrm{r})$
    $\pi \leftarrow \mathcal{P}((\mathsf{mpk}, \mathsf{ct}), (\mathrm{m}, \mathrm{r}), \mathsf{crs})$
    return $(\mathsf{ct}, \pi)$
On message GETPK from $\mathsf{B}$ to $G_\mathsf{att}$:
    return $\mathsf{spk}$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from $\mathsf{B}$ to $G_\mathsf{att}$:
    $\mathsf{eid} \leftarrow \{0,1\}^\lambda$
    $\mathcal{G}[\mathsf{eid}] \leftarrow (\mathsf{idx}, \mathsf{prog})$
    send $\mathsf{eid}$ to $\mathsf{B}$
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ from $\mathsf{B}$ to $G_\mathsf{att}$:
    if $\mathsf{input}[0] = \text{init-setup}$ then
        if $\mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_\mathsf{DE}$ then abort
        $(\text{init-setup}, \mathsf{eid}_\mathsf{KME}, \mathsf{crs}) \leftarrow \mathsf{input}$
        $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
    $\mathsf{output} \leftarrow (\mathsf{pk}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs})$
    $\sigma \leftarrow \mathcal{C}^{\Sigma}.\mathcal{O}_{\mathsf{sign}}(\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{output})$
    $\textbf{forward } (\mathsf{output}, \sigma) \text{ to } \mathsf{B}$
$\textbf{else if } \mathsf{input}[0] = \text{complete-setup } \textbf{then}$
    $\textbf{if } \mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_{\mathsf{DE}} \textbf{ then } \text{abort}$
    $(\text{complete-setup}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}) \leftarrow \mathsf{input}$
    $\mathrm{m} \leftarrow (\mathcal{G}[\mathsf{eid}][0], \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, \mathsf{ct}_{\mathsf{key}})$
    $\textbf{if } \Sigma.\mathsf{Vrfy}(\mathsf{spk}, \mathrm{m}, \sigma_{\mathsf{KME}}) \wedge \sigma_{\mathsf{KME}} \notin \mathcal{C}^{\Sigma}.\mathcal{Q} \textbf{ then}$
        $\text{output } (\mathrm{m}, \sigma_{\mathsf{KME}}) \text{ to } \mathcal{C}^{\Sigma}$
$\textbf{else if } \mathsf{input}[0] = \text{provision } \textbf{then}$
    $\textbf{if } \mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_{\mathsf{DE}} \textbf{ then } \text{abort}$
    $(\text{provision}, \sigma, \mathsf{eid}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathrm{F}}, \mathrm{F}) \leftarrow \mathsf{input}$
    $\mathrm{m}_1 \leftarrow (\mathcal{G}[\mathsf{eid}][0], \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, (\text{keygen}, \mathrm{F}))$
    $\textbf{if } \Sigma.\mathsf{Vrfy}(\mathsf{spk}, \mathrm{m}_1, \mathsf{sk}_{\mathrm{F}}) \wedge \mathsf{sk}_{\mathrm{F}} \notin \mathcal{C}^{\Sigma}.\mathcal{Q} \textbf{ then}$
        $\text{output } (\mathrm{m}_1, \sigma) \text{ to } \mathcal{C}^{\Sigma}$
    $\mathrm{m}_2 \leftarrow (\mathcal{G}[\mathsf{eid}][0], \mathsf{eid}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}], \mathsf{pk}_{\mathsf{DF}})$
    $\textbf{if } \Sigma.\mathsf{Vrfy}(\mathsf{spk}, \mathrm{m}_2, \sigma) \wedge \sigma \notin \mathcal{C}^{\Sigma}.\mathcal{Q} \textbf{ then}$
        $\text{output } (\mathrm{m}_2, \sigma) \text{ to } \mathcal{C}^{\Sigma}$
    $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{DF}}, \mathsf{msk}), \mathsf{output} \leftarrow (\mathsf{ct}, \mathsf{crs})$
    $\sigma_{\mathsf{DE}} \leftarrow \mathcal{C}^{\Sigma}.\mathcal{O}_{\mathsf{sign}}(\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{output})$
    $\textbf{forward } (\mathsf{output}, \sigma_{\mathsf{DE}}) \text{ to } \mathsf{B}$
$\textbf{else if } \mathsf{input}[0] = \text{run } \textbf{then}$
    $\textbf{if } \mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_{\mathsf{FE}[\mathrm{F}]} \textbf{ then } \text{abort}$
    $(\text{run}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \mathrm{y}') \leftarrow \mathsf{input}$
    $\textbf{if } \mathrm{y}' \neq \perp \textbf{ then}$
        $\mathsf{out} \leftarrow \mathrm{y}'$
    $\textbf{else}$
        $\mathrm{m} \leftarrow (\mathcal{G}[\mathsf{eid}][0], \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{crs})$
        $\textbf{if } \Sigma.\mathsf{Vrfy}(\mathsf{spk}, \mathrm{m}, \sigma) \wedge \sigma \notin \mathcal{C}^{\Sigma}.\mathcal{Q} \textbf{ then}$
            $\text{output } (\mathrm{m}, \sigma) \text{ to } \mathcal{C}^{\Sigma}$
        $\textbf{if } \mathsf{N}.\mathcal{V}((\mathsf{mpk}, \mathsf{ct}), \pi, \mathsf{crs}) = 0 \textbf{ then return } \perp$
        $\mathsf{mem} \leftarrow \mathsf{state}$
        $(\mathsf{output}, \mathsf{mem}') \leftarrow F(\mathsf{PKE.Dec}(\mathsf{msk}, \mathsf{ct}_{\mathsf{msg}}), \mathsf{mem})$
    $\sigma_{\mathsf{FE}} \leftarrow \mathcal{C}^{\Sigma}.\mathcal{O}_{\mathsf{sign}}(\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{out})$
    $\textbf{forward } (\mathsf{out}, \sigma_{\mathsf{FE}}) \text{ to } \mathsf{B}$
```

$\square$

Hybrid 3 Let Hybrid 3 replace the output of calls to the provision procedure for enclave KME with new value (ct',σ), where $\mathsf{ct}' \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{KD}, 0^{|\mathsf{sk}|})$ for the legitimate $\mathsf{pk}_{KD}, \mathsf{sk}$ held within the enclave, and σ is a valid attestation signature for an execution that produces ct'. If the PKE scheme internal to the KME program is CCA-secure, the two hybrids are indistinguishable to an attacker. Let Hybrid 3.1 replace the return value of calls to procedure provision on enclave DE with (ct',σ), with $\mathsf{ct}' \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{\mathsf{DF}}, 0^{|\mathsf{sk}|})$ and σ being a valid attestation signature on the produced output. Similarly, this hybrid is indistinguishable to the previous if PKE provides CCA security.

Below, we prove the following to lemmas, via reductions to CCA security of the encryption scheme. The two reductions are quite similar and depicted below.

Lemma 3.3. *If* PKE *is* CCA *secure, then* $\mathsf{H}_2 \approx \mathsf{H}_3$*, over the randomness used by* H_2*,* H_3.

Lemma 3.4. *If* PKE *is* CCA *secure, then* $\mathsf{H}_3 \approx \mathsf{H}_{3.1}$*, over the randomness used by those experiments.*

Proof. We use an adversary $\mathcal{Z}$ who can distinguish between the two hybrids H_2 and H_3 to construct an adversary $\mathbb{R}$ with the goal of breaking the CCA-security game challenger $\mathcal{C}^{\mathsf{PKE}}$. The challenge encryption ct replaces the encryption of pk_{KD}, the public key used to securely transfer the master secret key between the $\mathsf{prog}_{\mathsf{KME}}$ and $\mathsf{prog}_{\mathsf{DE}}$ enclaves. $\mathcal{Z}$ also instantiates a signature scheme Σ to reproduce the attestation role of G_{att}

Note: for this and all following reductions, we follow the convention that, for all subroutines (or parts of subroutines) not explicitly defined in the current reduction, the same code as the previous reduction applies. Furthermore, any calls to the challenge game is replaced with the corresponding primitive.

Reduction $\mathbb{R}^{\mathcal{Z},\mathcal{C}^{\mathsf{PKE}}}$

State variables	Description
pk_{KD}	Public key for game $\mathcal{C}^{\mathsf{PKE}}$
$\mathcal{H} \leftarrow \{\}$	Repository of message requests
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$
$\mathsf{state} \leftarrow \{\}$	Dictionary to hold the state for each function

```
*On message* $(\textsc{setup}, P)$:
  **if** $\mathsf{mpk} = \bot$ **then**
      $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
      $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
      **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
  **if** $P = \mathsf{A}$ **then**
      **send** $(\textsc{setup}, \mathsf{mpk})$ **to** $\mathcal{SC}_\mathsf{A}$
  **else if** $P = \mathsf{B}$ **then**
      **send** $(\textsc{setup}, \mathsf{mpk}, \mathsf{eid}_\mathsf{KME})$ **to** $\mathcal{SC}_\mathsf{B}$ and **receive** $(\textsc{provision}, \sigma, \mathsf{eid}_\mathsf{DE}, \mathsf{pk}_{KD})$
      $\mathrm{m}_0 \leftarrow \mathsf{msk}; \mathrm{m}_1 \leftarrow 0^{|\mathsf{msk}|}$
      **send** $(\textsc{challenge}, \mathrm{m}_0, \mathrm{m}_1)$ **to** $\mathcal{C}^\mathsf{PKE}$ and **receive** $\mathsf{ct}$
      $\sigma_\mathsf{sk} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, (\mathsf{idx}, \mathsf{eid}_\mathsf{KME}, \mathsf{prog}_\mathsf{KME}, \mathsf{ct}))$
      **send** $(\textsc{provision}, \mathsf{ct}, \sigma_\mathsf{sk}))$ **to** $\mathcal{SC}_\mathsf{B}$
*On message* $(\textsc{resume}, \mathsf{eid}, \mathsf{input})$ *from* $\mathsf{B}$ *to* $G_\mathsf{att}$:
  **if** $\mathsf{input}[0] = \text{init-setup}$ **then**
      **if** $\mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_\mathsf{DE}$ **then** abort
      $(\text{init-setup}, \mathsf{eid}_\mathsf{KME}, \mathsf{crs}) \leftarrow \mathsf{input}$
      $\mathsf{output} \leftarrow (\mathsf{pk}_{KD}, \mathsf{eid}_\mathsf{KME}, \mathsf{crs})$
      **forward** $(\mathsf{output}, \Sigma.\mathsf{Sign}(\mathsf{ssk}, (\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_\mathsf{KME}, \mathsf{output})))$
  **else**
      ...
*On message* $(\mathsf{Dec}, \mathsf{k}, \mathsf{c})$ *from* $\mathcal{Z}$ *to* $\mathsf{PKE}$:
  **if** $\mathsf{k} = \mathsf{sk}_{KD}$ **then**
      **return** $\mathcal{C}^\mathsf{PKE}.\mathsf{Dec}(\mathsf{sk}_{KD}, \mathsf{c})$
  **else**
      **return** $\mathsf{PKE.Dec}(\mathsf{k}, \mathsf{c})$
*On message* $(\textsc{output}, b)$ *from* $\mathcal{Z}$:
  output $b$
```

Let $\mathcal{Z}$ be an adversary that distinguishes between H_3 and $\mathsf{H}_{3.1}$; we construct an adversary $\mathbb{R}$ which calls onto $\mathcal{Z}$ to win the CCA game, by serving the challenge ciphertext to $\mathcal{Z}$ as pk_DF, the encryption key between enclaves running $\mathsf{prog}_\mathsf{DE}$ and $\mathsf{prog}_\mathsf{FE}$.

Reduction $\mathbb{R}^{\mathcal{Z}, \mathcal{C}^\mathsf{PKE}}$

State variables	Description
$\mathsf{pk}_{\mathsf{DF}}$	Public key for game $\mathcal{C}^{\mathsf{PKE}}$
$\mathcal{H} \leftarrow \{\}$	Repository of message requests
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$

```
On message $(\textsc{setup}, P)$:
  if $\mathsf{mpk} = \perp$ then
    $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
    $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
    send $\textsc{Get}$ to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
  if $P = \mathsf{A}$ then
    send $(\textsc{setup}, \mathsf{mpk})$ to $\mathcal{SC}_{\mathsf{A}}$
  else if $P = \mathsf{B}$ then
    send $(\textsc{setup}, \mathsf{mpk}, \mathsf{eid}_{\mathsf{KME}})$ to $\mathcal{SC}_{\mathsf{B}}$ and receive $(\textsc{provision}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$
    $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{KD}, \mathsf{msk})$
    $\sigma_{\mathsf{sk}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, (\mathsf{idx}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}}, \mathsf{ct}))$
    send $(\textsc{provision}, \mathsf{ct}, \sigma_{\mathsf{sk}}))$ to $\mathcal{SC}_{\mathsf{B}}$

On message $(\textsc{resume}, \mathsf{eid}, \mathsf{input})$ from $\mathsf{B}$ to $G_{\mathsf{att}}$:
  if $\mathsf{input}[0] = \text{init-setup}$ then
    if $\mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_{\mathsf{DE}}$ then abort
    $(\text{init-setup}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}) \leftarrow \mathsf{input}$
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
    $\mathsf{output} \leftarrow (\mathsf{pk}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs})$
    $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, (\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{output}))$
    forward $(\mathsf{output}, \sigma)$ to $\mathsf{B}$
  else if $\mathsf{input}[0] = \text{provision}$ then
    if $\mathcal{G}[\mathsf{eid}] \neq \mathsf{prog}_{\mathsf{DE}}$ then abort
    $\mathrm{m}_0 \leftarrow \mathsf{msk}; \mathrm{m}_1 \leftarrow 0^{|\mathsf{msk}|}$
    send $(\textsc{challenge}, \mathrm{m}_0, \mathrm{m}_1)$ to $\mathcal{C}^{\mathsf{PKE}}$ and receive $\mathsf{ct}$
    $\mathsf{output} \leftarrow (\mathsf{ct}, \mathsf{crs})$
    forward $(\mathsf{output}, \Sigma.\mathsf{Sign}(\mathsf{ssk}, (\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{output}))$ to $\mathsf{B}$
  else
    ...
```

□

Hybrid 4 This hybrid differs from $\mathsf{H}_{3.1}$ in how the Proof of Plaintext Knowledge is computed for a message encryption. Namely, instead of making calls to the honest prover, it simply creates simulated proofs using the trapdoor. The reduction defined below receives oracle access to either $\mathcal{P}$ or $\mathcal{S}_2$. Therefore, by distinguishing between the two experiments, one can break the zero-knowledge property of the NIZK scheme N. More formally, we prove the following lemma.

Lemma 3.5. *Assuming the zero-knowledge property of* N, $\mathsf{H}_{3.1} \approx \mathsf{H}_4$, *over the randomness used by those experiments.*

Proof. Given adversary $\mathcal{Z}$, which is capable of distinguishing between the two hybrids, we define an adversary $\mathbb{R}$, with the goal of breaking the zero-knowledge game $\mathcal{C}^{\mathsf{N}}$. The reduction against the zero-knowledge property of N is defined below.

Reduction $\mathbb{R}^{\mathcal{Z},\mathcal{C}^{\mathsf{N}}}$

State variables	Description
$\mathcal{H} \leftarrow \{\}$	Repository of message requests
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$
$\mathcal{J} \leftarrow \{\}$	Storage of N proofs
$\mathsf{state} \leftarrow \{\}$	Dictionary to hold the state for each function
$(\mathsf{crs}, \tau) \leftarrow \mathsf{N}.\mathcal{S}_1$	Simulated reference string and trapdoor

```
On message $(\textsc{setup}, P)$:
  if $\mathsf{mpk} = \perp$ then
    $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
    $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
  ...
On message $(\textsc{read}, \mathsf{h})$ from party $\mathsf{B}$ to $\mathcal{REP}$:
  $\mathrm{m} \leftarrow \mathcal{H}[\mathsf{h}]$; send $\textsc{Get}$ to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
  $\mathrm{r} \leftarrow \{0,1\}^\lambda$; $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m}; \mathrm{r})$
  send $(\textsc{challenge}, (\mathsf{mpk}, \mathsf{ct}), (\mathrm{r}, \mathrm{m}))$ to $\mathcal{C}^{\mathsf{N}}$ and receive $\pi$
  $\mathcal{J}[\mathsf{ct}_{\mathsf{msg}}] \leftarrow \pi$
  return $(\mathsf{ct}, \pi)$
On message $(\mathcal{P}, \mathsf{input})$ from $\mathcal{Z}$ to $\mathsf{N}$:
  send $(\textsc{challenge}, \mathsf{input})$ to $\mathcal{C}^{\mathsf{N}}$ and receive $\pi$
  return $\pi$
```

```
On message (OUTPUT, $b$) from $\mathcal{Z}$:
  output b
```

□

Hybrid 5 This hybrid is identical to H_4, but instead of executing the decryption of honestly generated ciphertexts, the decryptor enclave executes the extractor for the NIZK scheme to obtain the original plaintext message. This value is then encrypted by sending it to the ideal functionality, which stores it in its internal repository.

Lemma 3.6. *Assuming the extractability property of* N, $\mathsf{H}_4 \approx \mathsf{H}_5$, *over the randomness used by those experiments.*

Proof. Given an adversary $\mathcal{Z}$ capable of distinguishing between the two hybrids, we define an adversary $\mathbb{R}$ with the goal of breaking extractability game $\mathcal{C}^{\mathsf{N}}$.

Reduction $\mathbb{R}^{\mathcal{Z},\mathcal{C}^{\mathsf{N}}}$

State variables	Description
$\mathcal{H} \leftarrow \{\}$	Repository of message requests
$\mathcal{J} \leftarrow \{\}$	Storage of N proofs
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$
$\mathsf{state} \leftarrow \{\}$	Dictionary to hold the state for each function
$(\mathsf{crs}, \tau) \leftarrow \mathsf{N}.\mathcal{S}_1$	Simulated reference string and trapdoor

```
On message (RESUME, $\mathsf{eid}, \mathsf{input}$) from B to $G_{\mathsf{att}}$:
  if $\mathsf{input}[0] = \mathrm{run}$ then
    if $\mathcal{G}[\mathsf{eid}][1] \neq \mathsf{prog}_{\mathsf{FE[F]}}$ then abort
    $(\mathrm{run}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \mathsf{y}') \leftarrow \mathsf{input}$
    if $\mathsf{y}' \neq \perp$ then
      $\mathsf{out} \leftarrow \mathsf{y}'$
    else
      if $\Sigma.\mathsf{Vrfy}(\mathsf{spk}, (\mathcal{G}[\mathsf{eid}][0], \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{crs}), \sigma) = 0$ then return $\perp$
      if $\mathcal{J}[\mathsf{ct}_{\mathsf{msg}}] = \perp$ then
        $(\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
        if $\pi \in \mathcal{J}[(\cdot, \pi)]$ then send (DECRYPT, $F, \perp$) to B and abort
        $(\mathrm{m}, \mathrm{r}) \leftarrow \mathcal{E}(\tau, (\mathsf{mpk}, \mathsf{ct}), \pi)$
        if $\mathsf{N}.\mathcal{V}((\mathsf{mpk}, \mathsf{ct}), \pi, \mathsf{crs}) = 0 \vee \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m}; \mathrm{r}) \neq \mathsf{ct} \vee \mathsf{ct}_{\mathsf{msg}} \notin \mathcal{C}^{\mathsf{N}}.\mathcal{Q}$
  then
```

```
            output 1 to $\mathcal{C}^{\mathsf{N}}$
          $\mathsf{mem} \leftarrow \mathsf{state}$
          $(\mathsf{output}, \mathsf{mem}') \leftarrow \mathrm{F}(\mathrm{m}, \mathsf{mem}); \mathsf{out} \leftarrow \mathsf{output}$
        $\sigma_{\mathsf{FE}} \leftarrow \Sigma.\mathsf{Sign}(\mathsf{sid}, \mathsf{eid}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{out})$
        forward $(\mathsf{out}, \sigma_{\mathsf{FE}})$ to $\mathsf{B}$
  else
    ...
```

$\square$

Hybrid 6 This hybrid diverges from H_5 by replacing $\mathcal{REP}$'s copy of any message encrypted by A with an encryption of a string of zeros with the same length as the original plaintext message. Decryption is handled through the FESR functionality. The two hybrids can be distinguished by an adversary who can tell apart the two encrypted ciphertext, by winning the CPA security game.

Lemma 3.7. *Assuming CPA security,* $\mathsf{H}_5 \approx_{\varepsilon_{\mathsf{cca}}} \mathsf{H}_6$*, over the randomness used by those experiments.*

Proof. Given adversary $\mathcal{Z}$ who can distinguish between H_6 and H_5, we construct an adversary that can break CPA-security game $\mathcal{C}^{\mathsf{PKE}}$

Reduction $\mathbb{R}^{\mathcal{Z},\mathcal{C}^{\mathsf{PKE}}}$

State variables	Description
pk	Public key for game $\mathcal{C}^{\mathsf{PKE}}$
$\mathcal{G} \leftarrow \{\}$	Collects all enclave programs registered by $\mathcal{Z}$
$\mathcal{H} \leftarrow \{\}$	Repository of message requests

```
On message (SETUP, $P$):
  if $\mathsf{mpk} = \perp$ then
    $\mathsf{mpk} \leftarrow \mathsf{pk}$
    $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^{\lambda})$
    send GET to $\mathcal{CRS}$ and receive (CRS, $\mathsf{crs}$)
  ...
On message (READ, $\mathsf{h}$) from party $\mathsf{B}$ to $\mathcal{REP}$:
  $\mathrm{m}_0 \leftarrow \mathcal{H}[\mathsf{h}]; \mathrm{m}_1 \leftarrow 0^{|\mathrm{m}_0|}$
  send (CHALLENGE, $\mathrm{m}_0, \mathrm{m}_1$) to $\mathcal{C}^{\mathsf{PKE}}$ and receive $\mathsf{ct}, \mathrm{r}$
```

```
    $\pi \leftarrow \mathsf{N}.\mathcal{S}_2(\mathsf{crs}, \tau, (\mathsf{mpk}, \mathsf{ct}))$
    **return** $(\mathsf{ct}, \pi)$
  *On message* $(\mathsf{Enc}, \mathsf{k}, \mathrm{m})$ *from* $\mathcal{Z}$ *to* PKE:
    **if** $\mathsf{k} = \mathsf{mpk}$ **then return** $\mathcal{C}^{\mathsf{PKE}}.\mathsf{Enc}(\mathsf{pk}, \mathrm{m})$
    **else return** $\mathsf{PKE.Enc}(\mathsf{k}, \mathsf{ct})$
  *On message* $(\textsc{output}, b)$ *from* $\mathcal{Z}$:
    output $b$
```

□

Hybrid 7 In the last two hybrids we replace the encryption of the zero-strings (for the secret keys of the internal scheme) with the original keys. Therefore we have the following lemmas.

Lemma 3.8. *Assuming CCA security,* $\mathsf{H}_6 \approx_{\varepsilon_{\mathsf{cca}}} \mathsf{H}_7$*, over the randomness used by those experiments.*

Lemma 3.9. *Assuming CCA security,* $\mathsf{H}_7 \approx_{\varepsilon_{\mathsf{cca}}} \mathsf{H}_{7.1}$*, over the randomness used by those experiments.*

The reduction proceeds as in the parallel CCA hybrids (in the other direction) and is thus omitted.

Summary Note how $\mathsf{H}_{7.1}$ is in fact equivalent to the Simulator: we have shifted to an ideal world protocol (H_1) where the protocol can proceed only if the simulator verifies through attestation that the enclave programs are being run in the correct order (H_2) and we leak no information about inter-enclave secure channels ($\mathsf{H}_3, \mathsf{H}_{3.1}$). We then switch, through H_4 and H_5, from using genuine NIZK provers and verifiers (respectively) for honest parties into simulating the proof and extracting the witness (resp). By switching from encrypting messages to strings of zeros (H_6), we ensure no leakage is possible, while still using the original secrets for establishing secure channels ($\mathsf{H}_7, \mathsf{H}_{7.1}$). The final construction corresponds to our definition of the simulator in 3.4, and thus, through the subsequence of hybrids, protocol Steel UC emulates the ideal functionality FESR.

We now argue that all requirements of the UCGS theorem with respect to Steel, FESR and G_{att} are satisfied. In particular we prove the following lemma.

Lemma 3.10.

1. *The functionality* G_{att} *is a* FESR*-regular setup and subroutine respecting,*

2. FESR, Steel *are* G_{att}*-subroutine respecting,*

Proof. As required by [37, Definition 3.3], the global functionality G_{att} is FESR-regular, as does not invoke any new ITI of FESR and does not have an ITI with code FESR as subsidiary. G_{att} is clearly subroutine respecting. Also, FESR and Steel are G_{att}-subroutine respecting since they only make external calls to G_{att}. □

By the UCGS Theorem 2.3, Theorem 3.1 and the above lemma, UCGS security of the Steel protocol is concluded, i.e. for any parent protocol ρ which is $(\mathsf{Steel}, \mathsf{IDEAL}_{\mathrm{FESR}}, \xi)$-compliant and $(\mathsf{Steel}, M[x, G_{\mathsf{att}}], \xi)$-compliant for $x \in \{\mathsf{IDEAL}_{\mathrm{FESR}}, \mathsf{Steel}\}$, the protocol $\rho^{\phi \to \pi}$ UC-emulates ρ.

Update The proof in this chapter is preserved as it appeared in the published version of this work [53]. In the last stages of compiling this thesis, we became aware of a distinguishing attack between the real and ideal world. We discuss the nature of the attack and potential mitigations in Chapter 6.

Chapter 4

AGATE: Augmenting the modeling of Global Attested Trusted Execution

Computing is an endless cycle of inventing ways to isolate code in a private machine, followed by inventing ways to make it easier for those machines to interoperate

Carey Underwood

While the functionality modelled by G_{att} is a meaningful first step for modeling attested execution, in this chapter we will argue that it is too strong to capture realistic TEEs. As discussed in Section 2.2.3, there are many practical attacks that affect current implementations of TEEs, and the abstract G_{att} functionality of Pass, Shi, and Tramèr [230] we used in the previous chapter only models perfect enclave execution. Therefore, no TEE today can realise it (in a UC-emulation sense).

At a philosophical level, we will probably always live in a world where various TEE implementations will be available over time from different vendors, each with different capabilities. When a cryptographer or protocol designer begins working on their specification, they inevitably make certain assumptions about what features are required from their TEEs to support the correctness and security of their protocol. Previous works [278, 122] (as described in Section 2.2.4.1) have shown that, even if the G_{att} functionality is weakened, it is possible to prove security guarantees for certain protocols. If these models' assumptions are unrealistic given the capabilities of real TEE platforms at the time, the proofs will not be that informative for constructing deployable protocols in the real world. One way to rescue existing proofs might be augmenting a realistic TEE implementation with additional firmware and secondary protocols, to provide additional functionality or shield it from certain attacks, and showing the equivalence of the augmented TEE with a stronger setup.

To this end, this chapter provides a more careful treatment of trusted execution than the existing literature, focusing on the capabilities of enclaves and adversaries. Our goal is to provide meaningful patterns to show how different classes of TEEs compare to each other, in particular how a weaker TEE functionality can UC-emulate a stronger one, given an appropriate mechanism to bridge the two. We give a new, "modular" definition of TEEs, which captures a broad range of pre-existing functionalities defined in the literature, while maintaining the high level of abstraction of the previous chapter. While our goal is not directly to model implementations of specific commercial TEE providers, our modular definition provides a way to capturing more meaningful and realistic hardware capabilities. We provide a language to characterise how a class of TEEs

The chapter is structured as follows: in Section 4.1, we show that an overtly simplistic model of trusted execution can lead to protocols that can be proven secure but will fail in a real world scenario. This is exemplified by the introduction of rollback and forking attacks, which violate the security of Steel. We then propose, starting from Section 4.2, a generic framework for specifying Trusted Execution functionalities with granular enclave interfaces and adversarial powers. The key characteristic of our framework is to provide three parameters for each TEE setup: the set of features that an enclave running on the TEE can access; the set of attacks the adversary is allow to mount; and what enclave features are included in an attestation signature. In Section 4.3 , we provide a "Zoo" of enclave capabilities, adapting pre-existing formulations of enclaves into our model, as well as new capabilities that form useful building blocks for building protocols involving TEEs. Section 4.4 provides a notion of equivalence between different classes of TEEs, sketching a path to compiling programs designed for more powerful TEEs into programs that can run on weaker ones. Specifically, we use the feature and attack sets as parameters along which we can compare different types of TEE. We provide a generic compiler (and associated proof strategy) that shows how any TEE setup can be UC-emulated by the combination of a setup with fewer features, and a protocol that implements the missing one. Similarly, the combination of a setup with more attacks, and a protocol to defend against a portion of them, UC-emulate a setup with equivalent features, but without those attacks. Finally, we end the chapter with an illustrative example in Section 4.5, where we sketch how to rescue Steel security from rollback attacks through a simple protocol that relies on access to trusted storage.

Notation Unlike the previous chapter, our modelling in the next few sections will involve some UC-specific machinery, so we temporarily abandon some of the shorthands introduced in Section 2.1.2.3 to refer to some of the more low-level components of the framework.

As before, we use Interactive Turing machine Instances (ITI) to model any computation. Most of our formalisation in this chapter relies on structured protocols [75, Section 5.1]. A structured protocol is a list of nested ITIs, on which a higher level ITI (generally referred to as shell) has full access to read or overwrite the tapes of any lower level subroutine ITI (which we refer to interchangeably as the virtual ITI, or by their extended identities). ITIs have access to a number of tapes to store their identity, code, running memory, and communicate with other machines. Although the description of an ITI is not precise or prescriptive in terms of how it implements the computation, we assume that the program description uses some well-defined language, perhaps similar to a low level programming language or assembly. We represent each individual instruction as a command with optional arguments, which we represent using function call notation $\mathit{command}(\mathit{argument})$ sometimes with optional parenthesis (*command argument* e.g. for the case where *command* = **return**). We overload the set membership operator $\in$ to verify that the command component of the instruction belongs to the set. The code of an enclave can be seen as a list of ITI instructions of this type, and the notation "**for** instruction $\mathsf{i} \in \mathsf{prog}$ **do**" can be interpreted as iterating over the list of instructions for program prog (including command and arguments) without executing them (i.e. by advancing only the head of the shell over the tape). Conversely, when an ITI ρ in a structured protocol (see 2.1.2.1) contains pseudocode

```
begin executing input on $\pi$
for next instruction i on $\pi$ do $f(\mathsf{i})$
```

it should be read as ρ iterating through the code of a subroutine with extended identity π, and for each instruction i, ρ executes subroutine $f(\mathsf{i})$ to advance the state of π (updating its tapes and advancing π's head), while performing any additional operations in ρ's code.

4.1 Rollback and Forking Attacks

We now extend the G_{att} functionality to model *rollback* and *forking attacks* against an enclave, and show how this novel $G_{\mathsf{att}}^{\mathsf{rollback}}$ functionality does not allow securely running the Steel protocol.

Our model of rollback and forking attacks is partially inspired by the model of Matetic et al. [201], which distinguishes between enclaves and enclave instances. Enclave instances are independent copies of an enclave which share the same code but each maintain a distinct memory state. As with G_{att}, where the untrusted party has to call subroutines individually, the environment is not allowed to interact directly with a program once it is instantiated, except for pausing, resuming, or deleting enclave instances. Additionally, their model provides functions to store encrypted memory outside the enclave (*Seal*) and load memory back into an instance (*OfferSeal*).

In a typical rollback attack, an attacker crashes an enclave, erasing its volatile memory. As the enclave instance is restarted, it attempts to restart from the current state snapshot of sealed memory. By replacing this with a stale snapshot, the attacker is able to rewind the enclave state.

In a forking attack an attacker manages to run two instances of the same enclave in parallel, such that, once the state of one instance is changed by an external operation, querying the other instance will result in an outdated state. This is easily achieved on

TEEs where the attestation does not distinguish between two enclave instances running on the same machine. On a system where attestation uniquely identifies each enclave instance, a forking attack can still be launched by an attacker conducting multiple rollback attacks and feeding different stale snapshots to a single enclave copy [62].

Our new functionality $G_{\mathsf{att}}^{\mathsf{rollback}}$ employs some of these ideas to model the effect of both rollback and forking attacks. We replace the internal memory variable mem variable of G_{att} with a tree data structure, which stores the enclave's memory over time. The honest caller to the functionality will always continue execution from the memory state stored in an existing leaf of the tree. An adversary can instead mount a rollback attack by resuming execution from an arbitrary node (through a unique node identifier, which the adversary learns after each execution through message ITER). On a RESUME call that specifies such a node, the functionality loads the enclave state at that location before executing the input subroutine. The output of the computation mem' is then appended as a new child branch to the tree. After the rollback has taken place, the adversary can choose whether to continue the execution from this newly reset state (by calling resume without passing a valid node argument), execute a further rollback, or mount a forking attack by interactively choosing nodes in different branches of the tree. The functionality is parameterised with a signature scheme and a registry to capture all platforms with a TEE, like in the original formulation. We define algorithms $\mathsf{access}(\mathsf{t},\mathsf{n})$ to return the content of node labelled n in tree t; and $\mathsf{insertChild}(\mathsf{t},\mathsf{n},\mathsf{c})$ to return the new tree and node label resulting from inserting a child leaf with content c under node n in tree t ($\mathsf{insertChild}(\{\},\perp,\varepsilon)$ creates a new tree).

Functionality $G_{\mathsf{att}}^{\mathsf{rollback}}[\Sigma,\mathsf{reg},\lambda]$

State variables	Description
vk	Master verification key, available to enclave programs
msk	Master secret key, protected by the hardware
$\mathcal{T} \leftarrow \emptyset$	Table for installed programs

```
On message INITIALIZE from a party P:
    let $(\mathsf{spk},\mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda), \mathsf{vk} \leftarrow \mathsf{spk}, \mathsf{msk} \leftarrow \mathsf{ssk}$
On message GETPK from a party P:
    return $\mathsf{vk}$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
    **if** $P$ is honest **then assert** $\mathsf{idx} = P.\mathsf{sid}$
    generate nonce $\mathsf{eid} \xleftarrow{\$} \{0,1\}^\lambda$
    $(\mathsf{tree}, \mathsf{root}) \leftarrow \mathsf{insertChild}(\{\}, \perp, \varepsilon)$
    **store** $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}_, \mathsf{root}, \mathsf{tree})$
    **return** $\mathsf{eid}$
*On message* $(\text{RESUME}, \mathsf{eid}, \mathsf{input}, \mathsf{node})$ *from a party P where* $P.\mathsf{pid} \in \mathsf{reg}$:
    **let** $(\mathsf{idx}, \mathsf{prog}, \mathsf{lastnode}, \mathsf{tree}) \leftarrow \mathcal{T}[\mathsf{eid}, P]$, **abort** if not found
    **if** $P$ is honest **then let** $\mathsf{node} \leftarrow \mathsf{lastnode}$
    **let** $\mathsf{mem} \leftarrow \mathsf{access}(\mathsf{tree}, \mathsf{node})$
    **let** $(\mathsf{output}, \mathsf{mem}') \leftarrow \mathsf{prog}(\mathsf{input}, \mathsf{mem})$
    **let** $(\mathsf{tree}', \mathsf{child}) \leftarrow \mathsf{insertChild}(\mathsf{tree}, \mathsf{node}, \mathsf{mem}')$
    **if** $P$ is corrupted **then send** $(\text{ITER}, \mathsf{eid}, \mathsf{node}, \mathsf{child})$ **to** $\mathcal{A}$
    **let** update $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}, \mathsf{child}, \mathsf{tree}')$
    **let** $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{msk}, (\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output}))$ and **return** $(\mathsf{output}, \sigma)$
```

The proposed rollback model is perhaps somewhat simplistic, as it only allows "discrete" rollback operations (just as G_{att} allows descrete enclave resumes), where memory states are quantised by program subroutines. It is conceivable that real world attackers would have a finer-grained [68] rollback model, where they can interrupt the subroutine's execution, and resume from an arbitrary instruction. A limitation of the G_{att} (and $G_{\mathsf{att}}^{\mathsf{rollback}}$) model is that it does not provide us with such an instruction-level control over the program execution, meaning that at this level of abstraction it is impossible to model this type of attacks.

4.1.1 Rollback Attacks on Steel

Having introduced a new functionality with rollback and forking attacks, we are left with the question of whether the security statement for Steel (Conjecture 3.1) holds if we replace G_{att} with $G_{\mathsf{att}}^{\mathsf{rollback}}$. It is clear that we can not replace G_{att} functionality through the UC composition theorem, since $G_{\mathsf{att}}^{\mathsf{rollback}}$ can not UC emulate G_{att} - we can not construct a simulator to rollback G_{att} without changing its interface. Since the adversary is more powerful in the $G_{\mathsf{att}}^{\mathsf{rollback}}$-hybrid world, it might lead to attacks that are not otherwise possible in the G_{att}-hybrid world.

As a potential example, Yilek [306] presents a generic virtual machine rewinding attack on a program that uses an IND-CCA or IND-CPA secure encryption scheme. By applying repeated rollback attacks and running the encryption algorithm on multiple messages with the same randomness, the adversary can learn additional information on an encrypted message. It is not clear whether this attack would hold in our model, since the functionality does not explicitly state whether the random coins are stored in the memory being rewound. Since TEE platforms such as SGX usually have access to a hardware-based source of randomness [29], we assume (for now) that the attack is not possible, and that an enclave can access fresh randomness through the ITI random tape of the functionality. We will resume the discussion of this choice in the next section.

Instead of providing a generic attack, it is sufficient to show how Steel would be affected by the weaker functionality. Recall, from Chapter 3, that Steel is a protocol that UC-emulates ideal functionality FESR by letting the untrusted decryptor B evaluate stateful and randomised function F on encrypted data. B runs a decryption enclave (with code $\mathsf{prog}_{\mathsf{DE}}$) to fetch the master decryption key msk from authority C, and functional enclave $\mathsf{prog}_{\mathsf{FE}}[\mathrm{F}]$ to compute the function. $\mathsf{prog}_{\mathsf{DE}}$ shares a F-specific functional key with the functional enclave only if it has received evidence (through attestation) that it is running the right program.

Running Steel in the $G_{\mathsf{att}}^{\mathsf{rollback}}$-hybrid world would not allow a malicious B to learn anything about the state of function F (through the functional enclave). Enclave memory is still encrypted (sealed) in this setting, and the adversary only learns the labels and structure of the memory tree, not its contents. However, B can use its new rollback abilities to make Steel produce results that would not be possible in the G_{att}-hybrid world (or through the ideal functionality). As an example, take the following (stateful and randomised) function PRF-WRAPPER:

```
function PRF-WRAPPER(x, mem)
    if mem = $\emptyset$ then
        $\mathrm{K} \leftarrow \mathrm{x}$
        Store mem $\leftarrow \mathrm{K}$
        return ACK
    else if mem = $\perp$ then
        return $\perp$
    else
        Store mem $\leftarrow \perp$
        return $\mathrm{F}'(\mathrm{K}, \mathrm{x})$
```

PRF-WRAPPER implements a one-time Pseudo-Random Function: on its first call, it stores the input as a key; for the next call, it samples keyed PRF F'; thereafter it returns $\perp$.

An adversary who has:

1. completed initialisation of its decryption enclave with enclave id $\mathsf{eid}_{\mathsf{DE}}$;

2. obtained a functional key sk through the execution of keygen on $\mathsf{eid}_{\mathsf{KME}}$;

3. initialised a functional enclave for PRF-WRAPPER with enclave id $\mathsf{eid}_{\mathrm{F}}$, public key $\mathsf{pk}_{\mathsf{DF}}$, attestation signature σ, and leaked node id node for a leaf of the mem tree in $\mathsf{eid}_{\mathrm{F}}$;

can execute the following sequence of operations for three ciphertexts $\mathsf{ct}_{\mathrm{K}}, \mathsf{ct}_{\mathrm{x}}, \mathsf{ct}_{\mathrm{x}'}$, encrypting a key K and plaintexts x, x':

```
1: $((\mathsf{ct}_{\mathsf{key}}, \mathsf{crs}), \sigma_{\mathsf{DE}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\text{provision}, \sigma, \mathsf{eid}, \mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}))$
2: $((\text{computed}, ACK), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\text{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{k}}, \mathsf{crs}, \perp), \mathsf{node})$
3: // the adversary receives message $(\text{ITER}, \mathsf{eid}_{\mathrm{F}}, \mathsf{node}, \mathsf{node}')$ from $G_{\mathsf{att}}^{\mathsf{rollback}}$
4: $((\text{computed}, \mathrm{y}), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\text{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathrm{x}}, \mathsf{crs}, \perp), \mathsf{node}')$
5: $((\text{computed}, \mathrm{y}'), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\text{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathrm{x}'}, \mathsf{crs}, \perp), \mathsf{node}')$
6: // $\mathsf{node}'$ is the same node id as in the previous call (and thus to the parent
   of the current leaf in $\mathsf{mem}$)
```

If we visualise the function computed by PRF-WRAPPER as a finite state automaton, the adversary violates correctness as a result of this execution trace by inserting an illegal transition (with input ε) from state $\mathsf{access}(\mathsf{tree}, \mathsf{node}'.\mathit{child}) = \perp$ back to $\mathsf{access}(\mathsf{tree}, \mathsf{node}') = [K]$, and then back to state $\perp$ with input x'. The adversary can then obtain the illegal set of values $\mathrm{y} \leftarrow \mathrm{F}_{\mathrm{K}}(\mathrm{x})$ and $\mathrm{y}' \leftarrow \mathrm{F}_{\mathrm{K}}(\mathrm{x}')$, whereas in the ideal world where it is communicating with the ideal FESR functionality, after obtaining y the only possible output for the function would be $\perp$ (the only legal transition from state $\perp$ leads back to itself). The simulator is unable to address this attack, as the memory state is internal to the ideal functionality, and the key will always be erased after the second call.

One might think that on a second call for resume, the simulator could respond by sampling a value from the uniform distribution and feed it through the functional enclave's backdoor; however, the environment can reveal the key K and messages x, x' to the adversary, or conversely the adversary could reveal the uniform value to the environment. Thus the environment can trivially distinguish between the honest PRF output and the uniform distribution, and thus between the real and ideal world. Note that this communication between environment and adversary is necessary for universal composition as this leakage of $\mathrm{K}, \mathrm{x}, \mathrm{x}'$ could happen as part of a wider protocol employing functional encryption.

4.1.2 Rollback attacks on Iron

We also note that Stateless functional encryption as implemented in Iron [127] is resilient to rollback and forking because there is little state held between each enclave execution. Since the authority C is trusted, the only enclaves liable to be attacked are DE and FE[F].

DE stores PKE parameters after init-setup, and the decrypted master secret key after complete-setup. The adversary could try to gain some advantage by creating multiple PKE pairs before authenticating with the authority, but will never have access to the raw msk unless combining it with a leakage attack. Denial of Service is possible by creating concurrent enclaves (either DE or FE) with different public keys, and passing encrypted ciphertexts to the "wrong" copy which would be unable to decrypt (but it's not clear what the advantage of using rollback attacks would be, as the adversary could always conduct a DoS attack by denying the enclave access to any necessary system resources).

4.2 A modular G_{att} Setup

Motivation In the previous section, we extended the G_{att} functionality of [230] to provide rollback and forking attacks. While analysing the consequences of the weaker ideal functionality when used in conjunction with Steel, we remarked that the model does not account for how randomness is sampled. For that matter, the PST model does not really explain how enclave execution takes place in any detail. The abstraction of TEEs as an isolated execution mechanism with an easily verifiable proof of computation is a key insight of the model, and its promise of using the abstraction as a block box for constructing protocols a major selling point. However, as we have seen from the above example, the lack of detail can hamper reasoning about adversarial behaviour.

The formulation of G_{att} does not explicitly expose any TEE specific hardware or implementation details, beyond the abstract interface that allows the local party to install a program and execute it. When describing the components of G_{att} Pass, Shi, and Tramer [229, Section 3.2] explicitly state that the functionality emerges from a combination of the TEE features with some assumed firmware to provide this type of confidential computing service. In particular, they attribute the generation of unique per-enclave ids at installation, which are not guaranteed by all TEE architectures, to this firmware sampling a nonce from a unique key distributed to each TEE by the manufacturer during provisioning. A more careful approach would then consider the functionality provided by G_{att} as implementable by a combination of hardware, trusted firmware, and system-defined enclaves. The attestation signature guarantees that all of these components were acting in concert at the time when an output was generated.

Examining these components in more detail provides two advantages. First, it allows more meaningful relaxations of the security guarantees, by allowing to distinguish which components of the system can be compromised. Additionally, once we stop thinking of the functionality as a monolithic hardware component, it becomes natural to consider alternative features that the manufacturer or third parties might augment the TEE with. In particular, we may think of the combined hardware and software libraries an enclave has access to during its execution "runtime" as providing a kind of API. While the list of features provide by G_{att} could be considered a "standard" enclave interface, it is possible to imagine additional API calls available to the enclaves, for example a trusted clock [90], monotonic counters[90, 200], secure access to GPU compute resources [277, 290, 312] etc. Regardless of how these interfaces are implemented (e.g. by modifying the architecture or trusted firmware, or running the enclave through a "wrapper" library that interacts with a trusted system enclave, or even through a distributed protocol between multiple mutually untrusted enclaves), the attestation mechanism should capture their presence. Beyond showing that an enclave is running the correct program, a sound attestation mechanism also needs to certify to the verifier that the TEE provides the correct version of the API, otherwise the program code can not provide its security guarantees. In other words, a TEE functionality attests to the combination of (*prog, runtime*) rather than the mere application code *prog*.

Features, Attacks, and Attestation We now extend the G_{att} functionality from [230] (henceforth referred to as G_{att}^{PST}) to allow defining a larger class of TEE setups. Our goal is to capture the runtime behaviour of enclaves, without delving into the specifics of their implementation. To maintain this level of abstraction, we use a number of idealised interfaces.

Within our new formalism, a TEE application developer can choose to target a minimum set of features required bv their applications. A standard error will be returned if such a program is installed on an instance of the TEE functionality that does not support the feature set. For each possible modular formalisation of a TEE G_{att}^{mod}, we thus define a set of feature oracles $\mathbb{O}$, which represent the library of subroutines that are available to an enclave program. A feature of this kind is a polynomial time algorithm, as implemented by the runtime combination of hardware and software in that version of G_{att}^{mod}, including any communication with external parties. We also define a set of attack oracles $\mathbb{A}$ to capture adversarial behaviour. This can be thought as a parameter chosen by a protocol designer that captures "allowable" attacks in the current TEE setting under which the target protocol can still be proven secure. Any cryptographic protocol that wants to use TEE will therefore need to provide a lower bound for the set of required features $\mathbb{O}$, and an upper bound for the set of tolerated attacks $\mathbb{A}$, to parametrise their chosen version of G_{att}^{mod}. Relationships between different versions of TEEs are captured by the difference of these two sets, with equivalence statements made possible by running some additional runtime along enclave programs (either to increase the size of interfaces provided by $\mathbb{O}$, or to reduce the attacks available in $\mathbb{A}$).

We also introduce modularity in the attestation procedure. This is both to allow capturing a greater class of TEE architectures, as well as being a technical requirement. A reader familiar with the simulation framework will quickly realize that our programme of proving that, given the right runtime, a weaker TEE setup G_{att}^{mod} can UC-emulate the stronger $G_{\mathsf{att}}^{\prime mod}$, is hindered by the usage of a fixed signature scheme to model attestation. Since the two different TEE functionality would each sign different (*prog, runtime*) messages, it would be trivial for an environment to distinguish whether it is communicating with the real or ideal world. We therefore abstract the attestation mechanism in order to "program" the signature scheme.

Our model ties attestation and its verification to the specific G_{att}^{mod} functionality instance the user interacting with: the public parameters of the functionality allow a verifier to directly assess the capabilities of the attested enclave runtime and its adversary, and make an informed trust decision based on the feature and vulnerability of the enclave they are communicating with.

The functionality We now highlight the differences between the new formulation of G_{att}^{mod} (Fig. 4.1) and the original G_{att}^{PST} functionality (introduced in Section 2.2.4.1, and here reproduced in Fig. 4.2.

We iterate on the work of Section 3.4 to more carefully follow the conventions and formality of modern UC versions compared to G_{att}^{PST}. In particular, we now model enclaves as structured ITI subroutines to the G_{att}^{mod} functionality. On installation of an enclave, the functionality spawns a new ITI subroutine with composite extended iden-

Functionality $G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}]$

State variables	Description
$\mathsf{vk} \leftarrow \varepsilon$	Master verification key
$\mathsf{Sign} \leftarrow \varepsilon$	Attestation Signing algorithm
$\mathcal{S} \leftarrow \emptyset$	Table for signed messages
$\mathcal{T} \leftarrow \emptyset$	Table for installed programs

```
On message INITIALISE from a party P:
    send INITIALISE to $\mathcal{A}$ and receive $k, s$
    $\mathsf{vk} \leftarrow k, \mathsf{Sign} \leftarrow s$
On message GETPK from a party P:
    return vk
On message $(\text{VERIFY}, \sigma, m)$ from a party P:
    // Returns Boolean value
    return $m \in \mathcal{S}[\sigma]$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
    if pid is not corrupted then
        assert $\mathsf{idx} = \mathsf{sid}$
    for instruction $\mathsf{i} \in \mathsf{prog}$ do
        if $\mathsf{i} \notin \mathbb{O}$ then
            return MissingInstructionError
    generate nonce $\mathsf{eid} \xleftarrow{\$} \{0,1\}^{\lambda}$, store $\mathcal{T}[\mathsf{eid}, \mathsf{pid}] = (\mathsf{idx}, \mathsf{prog})$
    send INSTALL to $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
    return eid
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input}, \mathsf{attack})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
    let $(\mathsf{idx}, \mathsf{prog}) \leftarrow \mathcal{T}[\mathsf{eid}, \mathsf{pid}]$, abort if not found
    if $\mathsf{attack} = \varepsilon \vee$ pid is not corrupted then
        send input to $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ and receive output
    else
        assert $\mathsf{attack} \in \mathbb{A}$
        send $(\mathsf{attack}, \mathsf{input})$ to $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ and receive $\mathsf{output}, \mathsf{aux}$
        if $\mathsf{aux} \neq \varepsilon$ then
            query $\mathcal{A}$ with $(\mathsf{attack}, \mathsf{aux})$ and receive the reply CONTINUE
    let $\mathsf{meas} \leftarrow \mathbb{S}(\text{configuration of } \mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}])$
    let $\sigma \leftarrow \mathsf{Sign}(\mathsf{meas}), \mathcal{S}[\sigma] \leftarrow \mathcal{S}[\sigma]||\mathsf{meas}$
    return $(\mathsf{output}, \sigma)$
```

Figure 4.1: Global functionality G_{att}^{mod}

Functionality $G_{\mathsf{att}}[\Sigma, \mathsf{reg}, \lambda]$

State variables	Description
vk	Master verification key, available to enclave programs
msk	Master secret key, protected by the hardware
$\mathcal{T} \leftarrow \emptyset$	Table for installed programs

```
On message INITIALIZE from a party P:
    let $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda), \mathsf{vk} \leftarrow \mathsf{spk}, \mathsf{msk} \leftarrow \mathsf{ssk}$
On message GETPK from a party P:
    return $\mathsf{vk}$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
    if $P$ is honest then assert $\mathsf{idx} = P.\mathsf{sid}$
    generate nonce $\mathsf{eid} \overset{\$}{\leftarrow} \{0,1\}^\lambda$
    store $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}, \emptyset)$
    return $\mathsf{eid}$
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
    let $(\mathsf{idx}, \mathsf{prog}, \mathsf{mem}) \leftarrow \mathcal{T}[\mathsf{eid}, P]$, abort if not found
    let $(\mathsf{output}, \mathsf{mem}') \leftarrow \mathsf{prog}(\mathsf{input}, \mathsf{mem})$
    store $\mathcal{T}[\mathsf{eid}, P] \leftarrow (\mathsf{idx}, \mathsf{prog}, \mathsf{mem}')$
    let $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{msk}, (\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output}))$ and return $(\mathsf{output}, \sigma)$
```

Figure 4.2: The original G_{att}^{PST} functionality

tity[1] $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$, encoding the program prog, oracles $\mathbb{O}, \mathbb{A}$, the unique enclave ID eid, the identity pid for the party that installed the enclave, and the claimed session identity idx. The new subroutine is part of a UC *structured protocol*, where the top level subroutine with code $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$ spawned by G_{att}^{mod} is known as a *shell*, and a second subroutine with code prog created by the shell is known as the *body*. We use the shell of our structured protocol to capture modelling instructions related to the oracles, while the body is instantiated with the unaltered program code for the enclave (see Figure 4.3 for a graphical representation). Running enclaves as separate suboutine ITIs is functionally equivalent to running the input code within the global functionality as in the original treatment. It does provide, however, a cleaner abstraction, in that we are able to explicitly instantiate an ITI that runs the code of the enclave program installed, rather than having the ideal functionality act as an interpreter. In particular, our formalism now involves enclaves run by different parties being executed as separate ITIs, which we believe is a more natural model. Enclave programs are subroutine respecting in that the shell rejects any input message not sent through the G_{att}^{mod} functionality, and will only accept subroutine output messages from machines in its extended session. When resuming an enclave, the calling party might need to provide some additional import, depending on how much work the shell is required to carry out in addition to the enclave code execution in itself (e.g. if an enclave calls a feature that involves significant communication with external parties to be implemented, G_{att}^{mod} needs to be activated with sufficient import to activate those subroutines).

We parameterise each instance of G_{att}^{mod} by the static sets $\mathbb{O}, \mathbb{A}$ which capture feature and adversarial oracles respectively. On installation of a new enclave, G_{att}^{mod} first checks that all instructions in the proposed program code correspond to a call to one of the oracles in $\mathbb{O}$, and aborts with an error message if they are not. Both sets are the basis for the definition of the shell for all enclave subroutine ITIs installed by that instance of G_{att}^{mod}. We use the shell mechanism device to help us capture a specification of how the enclave program and the adversary can interact with the runtime. In particular, for each unique combination of oracles, we have to give a specific shell definition.

The shell detects when its enclave calls a feature oracle at runtime, and provides a return value. This can be derived through some local computation conducted by the shell, potentially after communicating with the adversary or other parties; or delegated

[1]recall that the identity of an ITI is made up of two strings: party ID and session ID. An extended identity combines the code for the ITI with the identity

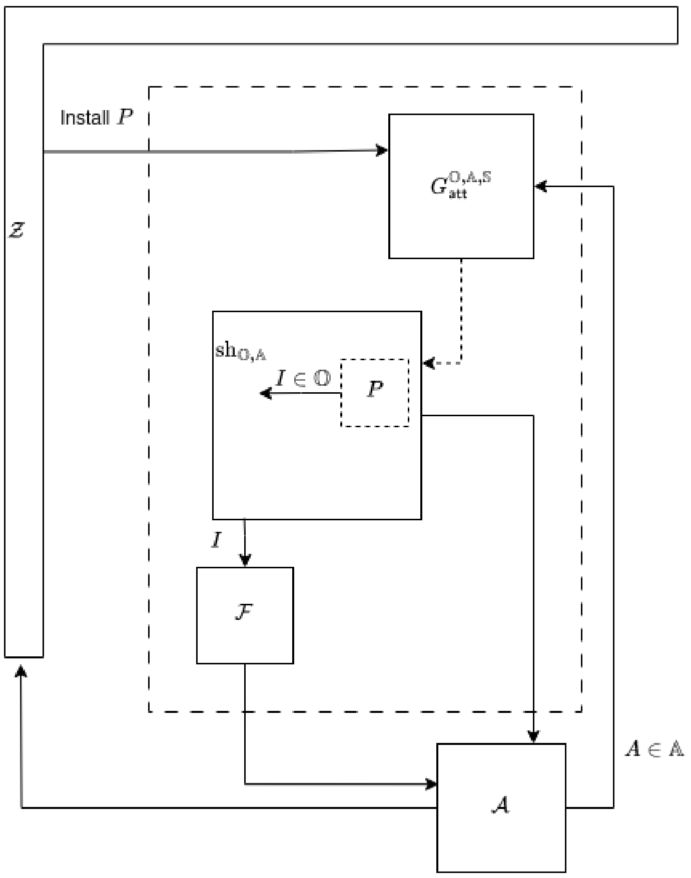

Figure 4.3: When a program with code P is installed on a G_{att}^{mod} enclave, the functionality spawns a new structured protocol subroutine with shell $\text{sh}_{\mathbb{O},\mathbb{A}}[]$ and body P. For some interfaces $I \in \mathbb{O}$, the shell will outsource its computation to some external functionality $\mathcal{F}$. The adversary $\mathcal{A}$ can interact with the enclave shell for any attacks $A \in \mathbb{A}$ through G_{att}^{mod}. Both $\text{sh}_{\mathbb{O},\mathbb{A}}[]$ and $\mathcal{F}$ can leak additional information to $\mathcal{A}$

to a distinct subroutine. When defining shell in this work we will generally use ideal subroutines, but this can be implemented through a real protocol without changing the definition (through UC-emulation).

A corrupted party is allowed to specify an auxiliary command along with their resume instructions that is executed by the shell in conjunction or instead of the normal program execution. The adversarial oracle is allowed to send a message to the adversary after the RESUME call has completed, and the adversary can in turn prevent the output of the program from being released with an attestation. The shell also handles any communication between enclaves that might be prompted by an attacker or feature oracle.

Finally, we parameterise the functionality by $\mathbb{S}$, a function that defines the contents of the attestation message for each enclave's execution. The original G_{att}^{PST} models an anonymous attestation signature scheme, and as such always produces an attestation signature tied to the set of arguments $(\mathsf{idx}, \mathsf{eid}, prog, \mathsf{output})$. This includes the claimed session ID for the current protocol executing the enclave, its unique enclave ID, the program code and the output of the most recent computation. Replacing this fixed data structure with a function allows us to model a broader range of attestation primitives, such as non-anonymous attestation (e.g. by including the UC party ID as one of the returned values, or a long-term public key tied to the party identity, as outlined in [229, Section 8.4]). We further relax the attestation mechanism of the G_{att}^{PST} functionality by allowing the adversary (through the simulator in the ideal world) to choose the format of attestation signatures, to allow the addition of details lacking in the high-level abstraction. Rather than having a full-fledged offline digital signature algorithm, the adversary provides (during the INITIALISE phase of the setup) G_{att}^{mod} with a public key and a signing algorithm s. The algorithm s is not required to be a well-formed signature scheme or guarantee typical security properties such as existential unforgeability. Therefore, G_{att}^{mod} implements signature verification by maintaining a map $\mathcal{S}$ of all signed strings and corresponding signatures generated by Sign. Verifications require sending a message to the setup, which checks whether it did produce the signed output through an "ideal" table lookup, rather than running a real verification algorithm as specified by the signature scheme. We still allow fetching a verification key for interface compatibility with G_{att}^{PST}, but any environment party that has obtained the relevant verification algorithm and key from the adversary will not have any guarantees of existential unforgeability.

When showing UC-emulation between two TEE setups, the simulator can provide a modified version of these algorithms to convince the environment that the ideal world TEE shares its runtime with the real world TEE. Take an adversary, for instance, that selects a signature scheme Σ, and initialises a G_{att}^{mod} instance with closure $s(\mathsf{meas}) = \Sigma.Sign(sk, \mathsf{meas})$, such that on a RESUME call, G_{att}^{mod} applies s to the value produced by function $\mathbb{S}$ over the configuration of the enclave ITI, the *enclave measurement*. On receiving algorithm s from the adversary, the ideal world simulator can derive a new $s'(\mathsf{meas}) = s(R(\mathsf{meas}))$. R is a transformation on the measurement that preserves all of its information, except that, if the measurement contains a public commitment to the program executed in the enclave (such as a hash of its source code), and the real world G_{att}^{mod} functionality is running code of type $\mathsf{prog} = (\mathsf{app}, \mathsf{runtime})$ for a specific runtime library, R replaces the commitment to enclave code app with a commitment to $(\mathsf{app}, \mathsf{runtime})$. This means that attestations in the ideal world will look like attestations to $(\mathsf{app}, \mathsf{runtime})$, despite G_{att}^{mod} only installing and executing app as part of its enclave. Of course, app still needs access to the service offered by the runtime, but in the ideal world it directly accesses the idealised features in the $\mathbb{O}$ set.

It is easy to show that G_{att}^{PST} UC-emulates G_{att}^{mod} for the sets of oracles and measurement function that correspond to G_{att}^{PST} (which we describe in the next section). We construct a simulator that selects the exact signature scheme specified in G_{att}^{PST}. Note that the opposite direction G_{att}^{mod} UC-emulates G_{att}^{PST} is more subtle. In fact, it is clear that the statement can not hold for all possible signature schemes provided by an adversary. Consider the null signature scheme where the signing algorithm $\mathsf{Sign}(\mathsf{ssk}, m) = 0^{\lambda}$; the signature scheme is still valid under the definition of G_{att}^{mod}, but it allows the environment to learn whether an enclave has produced a specific message, without having to communicate with it (by simply querying the ideal functionality for verification of an arbitrary measurement produced by $\mathbb{S}$). This is not possible in G_{att}^{PST}. A minimum entropy requirement for signatures provided by the adversary would therefore be necessary (but not sufficient) for the other direction of the equivalence.

A recent work by Canetti et al. [86] shows, as a corollary of the UCGS composition theorem, that if a global protocol G UC-emulates G' with respect to simulator S, then it is possible, in the general case of any context protocol ρ, to replace any subroutine call from ρ to G with a call to the combined subroutine of G' and S. This enables us to port any existing proofs that rely on G_{att}^{PST} (provided that the proof is valid under UCGS rather than GUC) into our new model, by simply replacing G_{att}^{PST} with the combination of the G_{att}^{mod} instance with equivalent $\mathbb{O}, \mathbb{A}$ oracles (which we describe in Section 4.3.1), and the simulator that at instantiation chooses the precise G_{att}^{PST} signature scheme over the usual $(\mathsf{idx}, \mathsf{pid}, \mathsf{prog}, \mathsf{output})$ measurement produced by $\mathbb{S}$.

4.3 Defining a G_{att}^{mod} Zoo

We now provide the definition for several sets of G_{att}^{mod} oracle instantiations, aiming to capture all existing variants of G_{att}^{PST} in the literature, as well as some natural extensions related to real world TEE realisations and attacks. Instantiating a shell for the functionality and adversarial oracles is a required step for using a new G_{att}^{mod} variant, and we have made efforts to write shells modularly so that they are easy to reuse. This does not mean that we can directly apply clean-room UC composition, but the structure of the shells makes it easy to mix and match them as required to handle additional oracles. In particular, most shells are structured around a loop that examines all instructions executed in the enclave subroutine ITI. When the instruction matches a specific oracle call, the shell shows how to implement it (in an ideal way). Some shells (such as the one presented in Section 4.3.4), modify the structure of ITIs created by the shell, but are still fully compatible with the formulation for the other shells.

For the remainder of this section, we consider versions of G_{att}^{mod} that use the same attestation signature function $\mathbb{S}$ as G_{att}^{PST} i.e. anonymous attestastions over $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output})$, unless stated otherwise .

4.3.1 G_{att}^{PST}

We begin by reformulating G_{att}^{PST} in the language of G_{att}^{mod}. While this is not made explicitly in the original work, G_{att}^{PST} relies on the following features:

- Addressable instructions: enclave execution begins at arbitrary instructions addressed through labels; in other words, the enclave program defines some entrypoint as functions/procedures/subroutines that can be called by the registerd party that installed the enclave, along with optional input arguments. On every execution, the enclave returns some output with an associated attestation signature

- Stateful resumes: each RESUME instruction is atomic, meaning that the subroutine will execute perfectly without any possibility for adversarial intervention. The state of the enclave is maintained across each sequential RESUME execution, and the adversary is not able to erase or otherwise tamper with it

- Sample Randomness: enclave programs are assumed to provide a true source of randomness (of arbitrary lengths)
- Unique Enclave Identifiers: a unique enclave ID is generated as a cryptographic nonce during enclave installation. Enclave IDs should be unique for all enclaves, regardless of which party installed them
- Attestation verification: attestation signatures can be verified from within the enclave program, without having to trust the external OS code to provide the attestation verification key as an input.

The first three notions are usually considered standard for Interactive Turing Machines. We therefore define the standard oracle set $\mathbb{O}^{\text{std}}$ to capture all ITI instructions that are standard for local computation. Although the operation of an Interactive Turing Machine are much more abstract, this can be thought of as the set of microarchitectural instruction provided by the processing unit executing the ITI. Attestation verification is explicitly not used in the G_{att}^{PST} paper[229, page 23], but we include it because many G_{att}^{PST}-hybrid protocols in the literature require the ability of verifying attestation inside an enclave. It could be argued that adding a capability to verify the attestation within an enclave makes the functionality less composable than intended, due to the inability to swap the fixed signature scheme with a call to an attestation service as provided by Intel for SGX. G_{att}^{mod} resolves this by moving verification to an abstract check in the functionality rather than verification of a concrete signature scheme.

We also note that the G_{att}^{PST} model forbids the enclave to have access to the UC PID for the party that is running it. While this is not explicitly stated, enclave programs with PID access could assist the party to establish a secure channel with another enclave-enabled party [229, Section 3.3].

As for the adversarial powers, even in the scenario where a host party is fully corrupted, adversarial interactions are limited when it comes to the PST enclaves. For any fully corrupted party, a G_{att}^{PST} adversary is able to install programs with arbitrary sessions identifiers under that host, honestly execute an enclave, and verify attestation signatures. These behaviours are all captured by default in the G_{att}^{mod} functionality, so no additional attack is required.

For capturing G_{att}^{PST} under G_{att}^{mod}, we thus define $\mathbb{O}$ to be the union of $\mathbb{O}^{\text{std}}$ and $\{\text{AttestVerify}\}$, and $\mathbb{A} = \{\}$. We now give an implementation for a UC shell that models enclave access to the oracle sets as defined. The extended identity of the shell is defined as $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“}\mathsf{att}\text{”}||\mathsf{idx}))$, where the PID is a concatenation of the enclave identifier generated by G^{mod}_{att} and the PID of the source machine which installed the enclave; the session SID is a concatenation of string *att* and the session of the protocol under which the enclave was installed. The enclave itself is a (virtual) subroutine ITI with extended identity $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$. While this is a simple shell, we examine it in detail, as it introduces patterns that are replicated in more complex shells later in this chapter.

```
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\mathrm{AttestVerify}\}$ and $\mathbb{A} = \{\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message INSTALL from $G^{mod}_{\mathsf{att}}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
On message input from $G^{mod}_{\mathsf{att}}$:
  begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  for next instruction i on virtual ITI do
    if $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ then
      allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
    else if $\mathsf{i} = \mathrm{AttestVerify}(\sigma, m)$ then
      send $(\mathrm{VERIFY}, \sigma, m)$ to $G^{mod}_{\mathsf{att}}$ and receive $v$
      append $v$ to subroutine output tape for virtual ITI
    else if $\mathsf{i} = (\textbf{return } v)$ then
      return $v$ with source $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
```

The shell receives message INSTALL when it is first created from G^{mod}_{att}, and it initialises the virtual ITI that will actually execute the enclave program. We make this step explicit in the pseudocode to mirror the interface of some of the shells presented later in the chapter, although it is not strictly necessary since UC creates a non-existing ITI when it first receive a message (if the `force − write` flag is set to 1).

Any other (non-INSTALL) input input the shell receives from G^{mod}_{att} must be the argument of a RESUME call, since the adversary is not able to give an attack message. Rather than writing input to the virtual ITI's input tape and letting it execute $\mathsf{prog}(\mathsf{input})$ directly, the shell observes the current configuration of $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$, and the instruction i that would be executed if it was activated with input (we denote this as “begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$”). With this the shell enters its main loop: depending on what type of instruction i is, it executes i following the specification in the appropriate branch, updates the configuration of $(\mathsf{prog},(\mathsf{eid},\mathsf{idx}))$, and chooses the next instruction.

The behaviour of this shell within the loop is fairly simple: most program instructions it considers will be in the standard oracle set $\mathbb{O}^{\mathrm{std}}$. In this case, the shell activates $(\mathsf{prog},(\mathsf{eid},\mathsf{idx}))$ with input i; as this is a simple instruction that any ITI can compute, the shell does not need to modify its behaviour, and it will allow the virtual ITI to execute it (updating its work tape) and immediately halt. The activation token now returns to the shell, which can select the next instruction i from the updated configuration.

When the instruction is of type $\mathrm{AttestVerify}(\cdot)$, the shell does not activate $(\mathsf{prog},(\mathsf{eid},\mathsf{idx}))$, but rather sends a message to G_{att}^{mod} to verify the attestation signature. Once it receives a boolean response, it writes it to the subroutine output tape of $(\mathsf{prog},(\mathsf{eid},\mathsf{idx}))$, and modifies the location of the tape head on its work tape. This essentially convinces the enclave virtual ITI that on its last activation it called the AttestVerify subroutine, and has just received its return value. We use this mechanism extensively in the rest of the section, as it allows modelling feature oracles so that the enclave program is oblivious of how they are computed.

Finally, when the next instruction i for the enclave is to return some value, the shell forwards it to G_{att}^{mod}, overwriting the sender-id of the outgoing message with its own extended identity. The shell thus yields activation back to G_{att}^{mod}, which proceeds with generating the attestation by calling $\mathbb{S}$ on the configuration of $(\mathsf{eid}||\mathsf{pid},\text{“att”}||\mathsf{idx})$.

4.3.2 Accessing a Clock

A natural extension of G_{att}^{PST}, which the original paper uses to realise fair MPC [229, Section 7.2], is to give the enclave access to a clock. The protocol is proven in a synchronous setting, where each party is activated in a round-robin fashion and is therefore aware of the round number. Enclaves are also equipped with round aware capabilities, even if they are not activated every round.

We now show how to realise a new G_{att}^{mod} functionality that supports feature oracles $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\mathrm{ReadRound}, \mathrm{IncRound}\}$ by giving it access to a local functionality that any protocol participant is allowed to interact with (both from within the enclave and outwith). Whenever the enclave program tries to execute an instruction interacting with the clock, the shell intervenes to forward the message to an ideal functionality, and inserts the value back into the enclave virtual ITI through the subroutine output tape.

```
                    $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\mathrm{ReadRound}, \mathrm{IncRound}\}$ and $\mathbb{A} = \{\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
  if ideal functionality $(\mathcal{F}_{\mathsf{clock}}, (\mathsf{idx}, \perp))$ does not exist, create
  send register to $\mathcal{F}_{\mathsf{clock}}$ on behalf of $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
On message input from $G_{\mathsf{att}}^{mod}$:
  begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  for next instruction i on virtual ITI do
    if $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ then
      allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
    else if $\mathsf{i} = \mathrm{ReadRound}$ then
      send READ to $(\mathcal{F}_{\mathsf{clock}}, (\mathsf{sid}, \perp))$ and receive $v$
      append $v$ to subroutine output tape for virtual ITI
    else if $\mathsf{i} = \mathrm{IncRound}$ then
      send INC to $(\mathcal{F}_{\mathsf{clock}}, (\mathsf{sid}, \perp))$ and receive $v$
      append $v$ to subroutine output tape for virtual ITI
    else if $\mathsf{i}$ =(return $v$) then
      return $v$ with source $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
```

The INSTALL subroutine of this shell installs the virtual ITI for a new enclave, and ensures that an instance of the ideal functionality for the clock exists in this session (with a standard PID $\perp$). It then sends a registration message for the enclave to $\mathcal{F}_{\mathsf{clock}}$. For enclave RESUME calls, the structure of the shell execution loop is the same as in the shell from last section, with the instructions executed by the enclave for either ReadRound, IncRound oracle calls forwarded to the ideal functionality, and its return values returned to the enclave in the same way that we added the return value for an attestation verification call in the previous section. We now describe the behaviour of the clock functionality

Functionality $\mathcal{F}_{\mathsf{clock}}$

```
The identity of the functionality is $(\mathsf{sid}_F, \perp)$
On message REGISTER from a party P:
  if t = {} then $r \leftarrow 0$
  if $P.\mathsf{sid} = \mathsf{sid}_F$ then
    $t[\mathsf{pid}] \leftarrow \perp$
On message READ from a party P:
  return $r$
On message INC from a party P:
  if $P.\mathsf{sid} = \mathsf{sid}_F$ then $t[P.\mathsf{pid}] \leftarrow \top$
    if all values in $t = \top$ then
      $r$++
      reset all values in $t$ to $\perp$
    return $r$
```

$\mathcal{F}_{\mathsf{clock}}$ provides a per-session round counter functionality. A round is increased when all registered parties consent to. Internally, it stores the round counter as a monotonically increasing integer r, and records whether a party has agreed to increase the round via dictionary t, which records a boolean value for each party. Once a party sends an INC message, they are not allowed to withdraw. After the last registered party agrees to increase, r is incremented, and all values in t are set to false. A new part can register at any point, and the value of the round counter is publicly accessible.

4.3.3 Interrupting computation

As a first attempt to show how to capture an attack oracle, we now model a new version of G_{att}^{mod} where enclave programs are explicitly able to control which objects in their memory can be saved to confidential persistent storage. An enclave is able to preserve state across enclave executions by *storing* arbitrary bitstrings in an encrypted form, and later *fetch* it back into memory when next resumed. Only the original enclave itself is able to access any data it stored through the oracle call; the adversary only learns the size of what was stored. In Intel SGX, these features are known as sealing and unsealing.

As the enclave now interacts with the (untrusted) memory of the host, the adversary will be notified of any storage or fetching attempt, and will have a chance to censor them. Given that the program integrity relies on these external oracle calls completing, this is equivalent to the adversary aborting the enclave program. We therefore provide the adversary with oracles $\mathbb{A} = \{\text{Abort}, \text{Continue}\}$. The adversary can stop a memory access oracle from completing, but can not erase or leak external memory that was already successfully stored. This example oracle combination for G_{att}^{mod} is for illustrative purposes; a more realistic oracle definition would let memory operations return to the enclave with an error, allowing the program execution to continue, and allow the adversary to permanently erase external memory.

We define the following shell:

$\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\text{std}} \cup \{\text{Store}, \text{Fetch}\}$ and $\mathbb{A} = \{\text{Abort}, \text{Continue}\}$

The extended identity of the shell is defined as $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$

State variables	Description
$mem \leftarrow \varepsilon$	Persistent memory storage for the enclave

```
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
  set $\mathsf{halt} \leftarrow \bot$
On message input from $G_{\mathsf{att}}^{mod}$:
  if $\mathsf{halt} = \top$ then abort
  begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  for next instruction i on virtual ITI do
    if $\mathsf{i} \in \mathbb{O}^{\text{std}}$ then
      allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
    else if $\mathsf{i} \in \{\text{Store}(s), \text{Fetch}\}$ then
      if pid is corrupted then
        $\mathsf{halt} \leftarrow \top$
        Send message $(\text{STORE}, |s|) \vee \text{FETCH}$ to $\mathcal{A}$ and await
        if next message on the input tape is Abort from $G_{\mathsf{att}}^{mod}$ then
          erase work tape contents of virtual ITI and return
        else if next message on the input tape is Continue from $G_{\mathsf{att}}^{mod}$ then
          $\mathsf{halt} \leftarrow \bot$
      if $i = \text{Store}(s)$ then
```

```
            $mem \leftarrow s$
        **else if** $i =$ Fetch **then**
            append $mem$ to subroutine output tape for $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    **else if** $\mathsf{i} = (\textbf{return}\ v)$ **then**
        **return** $v$ with source $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
*On message* Abort *from* $G_{\mathsf{att}}^{mod}$*:*
  **if** $\mathsf{halt} = \perp$ **then**
    set $\mathsf{halt} \leftarrow \top$
    erase work tape contents of virtual ITI and **return**
*On message* $(\text{Continue}, \mathsf{input})$ *from* $G_{\mathsf{att}}^{mod}$*:*
  **if** $\mathsf{halt} = \perp$ **then**
    parse $(cmd, args) \leftarrow \mathsf{input}$
    **return** $cmd(args)$
```

Unlike the previous two shells, the execution loop of the above includes adversarial interactions as part of the enclave operation. In particular, when an enclave run by a corrupted party tries to interact with external memory by calling a Store or Fetch instruction, the shell sets flag $\mathsf{halt} \leftarrow \top$, notifies the adversary, and relinquishes the activation token. On its next activation, if it finds a message from the set $\mathbb{A}$, it resumes execution from where it last stopped. Otherwise, on any other input, it will abort (as long as flag $\mathsf{halt} = \top$): storing and fetching are *blocking*.

The adversary $\mathcal{A}$ only learns that enclave eid run by party pid in session idx is either trying to read from external storage, or that is writing some data and its size. $\mathcal{A}$ replies by sending a message of type $(\text{RESUME}, \mathsf{eid}, \varepsilon, a \in \mathbb{A})$ from corrupted party pid to G_{att}^{mod}. If $a =$ Continue, the shell continues executing from where it left off, storing bitstring s “ideally” (within its own internal variable *mem*). Otherwise, if $a =$ Abort, the enclave crashes, losing all memory stored within the virtual ITI’s work tape. An Abort attack is not final: depending on the code of prog, the enclave can be resumed later on, and recover some partial state from the last value successfully stored to mem, if any. The $\mathcal{A}$ can call the attack oracles at any other point, without the enclave trying to access memory (i.e. when $\mathsf{halt} = \perp$); on an Abort call, the shell erases the enclave’s working memory as well; on a Continue call, the shell simply executes the provided argument as a resume operation.

Within the above definition, the shell variable halt keeps track on whether the adversary has instructed the enclave to stop. On every call to Store or Fetch, the shell yields to the adversary, informing it on what type of instruction the enclave has requested, including the length of the message that's being stored (but not the contents, memory storage is still confidential). These requests are blocking, so we do not allow any other enclave operation to be executed until the adversary replies with a Abort or Continue on the input tape. On an Abort message, the current resume execution is halted, and any memory in the enclave's worktape is erased. If the adversary instead issues a Continue message (with no arguments), the enclave will resume from where it stopped. If the adversary issues a Abort followed by a Continue, it should pass an argument to an appropriate subroutine of the program, which might Fetch whatever memory was last stored to let the program recover from a last known state.

4.3.4 Rollback Attacks

While the previous version of G_{att}^{mod} describes an adversary that is able to stop an enclave from storing any data to an external medium, the integrity and freshness of a successfully stored message is always guaranteed by a successful Fetch. We now explore a model with a slightly stronger adversary, who controls the storage medium and can overwrite the external memory location. Despite this, the enclave will not accept arbitrary messages, but only ones that were produced during a legitimate Store operation.

In Section 4.1, we introduced a new variant of G_{att} that allows state continuity attacks, $G_{\mathsf{att}}^{\mathsf{rollback}}$. Recall that $G_{\mathsf{att}}^{\mathsf{rollback}}$ tracks enclave state updates in a tree-like structure, and allows the adversary to specify an index for an arbitrary node in the tree to resume enclave execution from a specific snapshot. The tree allows the adversary to fork the enclave at an arbitrary state and maintain multiple copies that can progress independently.

As we no longer track the state of an enclave in a table $\mathcal{T}$, an instance of G_{att}^{mod} that supports Rollback or Fork instructions in $\mathbb{A}$ will require an alternative mechanism to maintain the state. We implement this through an enclave shell that executes each RESUME operation as a distinct virtual ITI. After the RESUME returns, the shell instantiates a new ITI by copying the last active configuration, and notifies the adversary of a unique pointer for that execution through an ITER message. When the adversary calls for a Rollback or Forking attack with a specific pointer, the shell can run the provided input on with adequately stale state by activating the older ITI that the pointer corresponds to.

```
                         $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}}$ and $\mathbb{A} = \{\mathrm{Rollback}, \mathrm{Fork}\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
  generate nonce $c \xleftarrow{\$} \{0,1\}^{\lambda}$
  create virtual ITIs $(\mathsf{prog}, (\mathsf{eid}||\emptyset, \mathsf{idx})), (\mathsf{prog}, (\mathsf{eid}||c, \mathsf{idx}))$
  if pid is corrupted then send $(\textsc{Iter}, \emptyset, c)$ to $\mathcal{A}$
On message input from $G_{\mathsf{att}}^{mod}$:
  execute input on virtual ITI $(\mathsf{prog}, (\mathsf{eid}||c, \mathsf{idx}))$
  generate nonce $c' \xleftarrow{\$} \{0,1\}^{\lambda}$
  copy working tape of $(\mathsf{prog}, (\mathsf{eid}||c, \mathsf{idx}))$ into new virtual ITI $(\mathsf{prog}, (\mathsf{eid}||c', \mathsf{idx}))$
  if pid is corrupted then send $(\textsc{Iter}, c, c')$ to $\mathcal{A}$
  $c \leftarrow c'$
On message $(\textsc{Rollback}, (i, \mathsf{input}))$ from $G_{\mathsf{att}}^{mod}$:
  execute $(\mathsf{out}, (\textsc{Fork}, c, i, i')) \leftarrow (\textsc{Fork}, i, \mathsf{input})$
  $c \leftarrow i'$
  return $(\mathsf{out}, (\textsc{Rollback}, i, i'))$
On message $(\textsc{Fork}, (i, \mathsf{input}))$ from $G_{\mathsf{att}}^{mod}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}||i, \mathsf{idx}))$ exists then
    $\mathsf{out} \leftarrow \varepsilon$
    if $\mathsf{input} \neq \varepsilon$ then
      execute input on $(\mathsf{prog}, (\mathsf{eid}||i, \mathsf{idx}))$, read subroutine output tape into out
    generate nonce $i' \xleftarrow{\$} \{0,1\}^{\lambda}$
    copy work tape of $(\mathsf{prog}, (\mathsf{eid}||i, \mathsf{idx}))$ to $(\mathsf{prog}, (\mathsf{eid}||i', \mathsf{idx}))$
    return $(\mathsf{out}, (\textsc{Fork}, c, i, i'))$
```

The structure of each subroutine's extended identity involves appending a unique pointer nonce to the enclave id (the initial state is denoted by special pointer $\emptyset$). Variable c holds the pointer to the latest snapshot of the enclave accessible by a honest RESUME command. After each honest execution, the enclave creates a new UC subroutine by generating a new id and copies the execution tape of the subroutine c points to into this new copy, which is where the new instructions will be executed. The adversary always learn the pointer generated for each iteration. If the adversary conducts a Rollback (by sending message ($\textsc{Resume}, \mathsf{eid}, \mathsf{input}, \mathrm{Rollback}$) from corrupted party pid to G_{att}^{mod}), c is overwritten with the pointer for an ITI whose memory state is copied from the one the adversary provided a pointer for. In a fork, c is not affected, but the adversary learn of new pointer i' it can access. In both cases, the shell returns to G_{att}^{mod} with the enclave output (if the attack also contained an instruction to execute) and auxiliary information on the new ITI pointer. Since the attack was successful, G_{att}^{mod} waits for the adversary to issue a CONTINUE message to finalise the return value and produce attestation (otherwise the RESUME call for G_{att}^{mod} never terminates).

It is clear from our formulation that a rollback is just a special case of a fork, where one of the two fork branches is not used again (in fact, on any ROLLBACK message, the shell executes the FORK subroutine with the appropriate parameters). Distinguishing the two cases is primarily useful in the setting of a mobile adversary. While corrupted, a party can always choose the index for an enclave copy it wants to execute through the FORK command. When the party is no longer corrupted, however, the only copy of the enclave that can be executed is the one at index c. The adversary can thus use ROLLBACK to force the newly honest party to execute the enclave from an arbitrary state, essentially erasing the access to any state that might have succeeded it.

4.3.5 Modelling side channels

As we have discussed in the Background chapter 2.2.4.1, some previous works in the literature have extended the G_{att}^{PST} model to capture additional types of side-channel attacks. We now adapt those extensions into G_{att}^{mod} shells.

Transparent enclaves Tramer et al. [276] provides a (local) UC functionality for attested execution with no confidentiality guarantees, later extended in [229, Section 8] to the global setting. Enclaves in this Transparent Enclave setting suffer from leakage of all internal memory, except for the master signing key for attestation. This allows integrating an enclave with such a leakage in protocols that only require the integrity provided by enclaves. The modeling of transparent enclave is a simple extension over that of G_{att}^{PST}: the output of each resume call is followed by the leakage of the random bits sampled by the enclave program. Knowing the inputs, randomness and the code of the program is sufficient to reconstruct its operation and internal memory for any randomised program, whereas deterministic programs are inherently transparent by default, since the adversary knows the code of the enclave when they install it.

In the language of G_{att}^{mod}, we state that for any attested functionality with $RandomSample \in \mathbb{O}$ (and therefore any adversary where $\mathbb{O}^{\text{std}} \subset \mathbb{O}$), we can realise a transparent version by including TransparentLeak in the adversarial oracles $\mathbb{A} =$. We recover the modelling from [276] and [229, Section 8.1] by letting the shell leak produce the entirety of the virtual ITI random tape to the adversary after each execution. On installation, enclaves start in the default non-transparent state, but once the adversary issuses the TransparentLeak attack, all further values of the tape are leaked.

```
                         $\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\text{std}}$ and $\mathbb{A} = \{\text{TransparentLeak}\}$
The extended identity of the shell is defined as $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
*On message* INSTALL *from* $G_{\mathsf{att}}^{mod}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
  $\mathsf{transparent} \leftarrow \bot$
*On message* input *from* $G_{\mathsf{att}}^{mod}$:
  begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  **for** next instruction i on virtual ITI **do**
    **if** $\mathsf{i} \in \mathbb{O}^{\text{std}}$ **then**
      allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
    **else if** i =(**return** $v$) **then**
      **if** $\mathsf{transparent} = \top \wedge \mathsf{pid}$ is corrupted **then**
        **send** $(\text{LEAK}, \text{random tape of } (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$ **to** $\mathcal{A}$
      **return** $v$ with source $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
*On message* $(\text{TRANSPARENTLEAK}, \mathsf{input})$ *from* $G_{\mathsf{att}}^{mod}$:
  set $\mathsf{transparent} \leftarrow \top$
  **if** $\mathsf{input} \neq \varepsilon$ **then**
    parse $(cmd, args) \leftarrow \mathsf{input}$, **return** $cmd(args)$
  **else**
    **return** $(\varepsilon$, random tape of $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$
```

A stronger type of leakage would leak the entirety of the virtual ITI's work tape. This would allow the adversary to recover any shared secret that predate the corruption attack. This can be implemented by simply appending the work tape to the LEAK message, or allow the adversary to apply standard UC passive corruption to the virtual

ITI.

Almost-transparent and Semi-honest enclaves Dörre, Mechler, and Müller-Quade [122] introduce two relaxations over the G_{att} functionality that aim to capture a middle ground between the side-channel free G_{att}^{PST} and transparent enclaves. Their models provides enclaves with access (in our language) to feature oracles for secure key exchange and symmetric encryption.

We now provide an implementation for a shell that implement these cryptographic functions by outsourcing them to local functionality $\mathcal{F}_{\mathsf{crypto}}$ as defined by Küsters and Rausch [172].

$\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\mathrm{KeyExchange}, \mathrm{SKEGen}, \mathrm{SKEEnc}, \mathrm{SKEDec}, \mathrm{ReleaseKey}\}$
and $\mathbb{A} = \{\mathrm{TransparentLeak}, \mathrm{Halt}\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$

State variables	Description
$\mathcal{E} \leftarrow \{\}$	Stores Group elements received by other enclaves

```
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
    if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
    if ideal functionality $(\mathcal{F}_{\mathsf{crypto}}, (\mathsf{idx}, \perp))$ does not exist, create
    send GETDHGROUP to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{DHGROUP}, G, n, g)$
On message input from $G_{\mathsf{att}}^{mod}$:
    if $\mathsf{halt} = \top$ then abort
    begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    for next instruction i on virtual ITI do
        if $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ then
            allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
        else if $\mathsf{i} = (\mathrm{KeyExchange}, \mathsf{pid}', \mathsf{eid}')$ then
            set $\mathsf{halt} \leftarrow \top$
            send GENEXP to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{EXPOPOINTER}, ptr^e, g^e)$
            if pid is corrupted then
                query $\mathcal{A}$ with $(\mathrm{KEYEXTO}, \mathsf{pid}', \mathsf{eid}')$ and receive the reply continue
            if $\mathcal{E}[\mathsf{pid}', \mathsf{eid}'] = \perp$ then
            // no stored keyshare for eid', we are the initiatior
            send $(\textsc{KeyEx}, g^e)$ to $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid'}||\mathsf{pid'}, \text{“att”}||\mathsf{idx}))$ and await
            while next message on the input tape is not $(\textsc{KeyEx}, \mathsf{pid'}, \mathsf{eid'}, h)$ do ignore
            send $(\textsc{BlockGroupElement}, h)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive OK
        else
            // eid' was the key exchange initiatior, we already have $h$
            $h \leftarrow \mathcal{E}[\mathsf{pid'}, \mathsf{eid'}]$
        send $(\textsc{GenDHKey}, ptr^e, h)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Pointer}, ptr^{dhk})$
        send $(\textsc{Derive}, ptr^{dhk}, \texttt{unauth-key})$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Pointer}, ptr^{sk})$
        set halt $\leftarrow \bot$
        append $ptr^{sk}$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = \text{SKEGen}$ then
        send $(\textsc{New}, \texttt{unauth-key})$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Pointer}, ptr)$
        append $ptr$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\text{SKEEnc}, ptr, m)$ then
        send $(\textsc{Enc}, ptr, m)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Ciphertext}, hdl)$
        append $hdl$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\text{SKEDec}, hdl, ct)$ then
        send $(\textsc{Dec}, ptr, ct)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Plaintext}, m)$
        append $m$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\text{ReleaseKey}, ptr)$ then
        send $(\textsc{Retrieve}, ptr)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\textsc{Key}, k)$
        append $k$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} =$(return $v$) then
        return $v$ with source $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message HALT from $G_{\mathsf{att}}^{mod}$:
  set halt $\leftarrow \top$
  return
On message $(\textsc{KeyEx}, h)$ from $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid'}||\mathsf{pid'}, \text{“att”}||\mathsf{idx}))$:
  if halt $= \bot$ then
    // we are not waiting for key exchange to complete;
    // eid' is the initiator
    send $(\textsc{BlockGroupElement}, h)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive OK
    $\mathcal{E}[\mathsf{pid'}, \mathsf{eid'}] \leftarrow h$
  // if the enclave is halted, eid is the initiator; on message KeyEx,
  we exit the loop to complete the key exchange
```

Most of the oracle calls in the shell are simply forwarded from the enclave to the ideal functionality. KeyExchange is more interesting, as it is our first oracle call that involves direct communication between two enclaves. We implement a "synchronous" key exchange, in that we expect both enclaves to call the respective KeyExchange oracle to establish a channel. We do not provide a mechanism for enclaves to discover enclave IDs, and assume that they are provided by one of the other protocol inputs. The first enclave to call the oracle will stop accepting any further activations until the key exchange protocol completes (we refer to this enclave as the initiator). If the other enclave's shell receives a KEYEX message before its enclave has reached the KeyExchange call, it will store the received share dictionary $\mathcal{E}$ to be retrieved at a later point. Once both parties have communicated their shares to each other, the shared key is computed by the $\mathcal{F}_{\mathsf{crypto}}$ functionality. Rather than returning it directly to the two enclaves, our shell uses it to derive a new symmetric key, which is what is obtained by both parties as the return value of KeyExchange (this step is necessary because $\mathcal{F}_{\mathsf{crypto}}$ does not allow using keys of type `dh-key` for symmetric operations). If either party running the enclave is corrupted, the adversary can learn that the key exchange is taking place and issue a HALT message. Additionally, the adversary might learn any other information leaked by F and its leakage functions.

The addition of these oracles does not provide the enclave with meaningful new capabilities on its own, since an enclave can implement these operations as part of a library with access to randomness and attestation verification. However, it becomes significant once it is combined with the TransparentLeak attack: by executing the secure operations "ideally " through oracles, the randomness needed to compute them is not leaked as part of the transparent attack. Dörre, Mechler, and Müller-Quade [122] define an enclave with access to both $\{\text{KeyExchange}, \text{SKEGen}, \text{SKEEnc}, \text{SKEDec}, \text{ReleaseKey}\} \in \mathbb{O}$ and TransparentLeak $\in \mathbb{A}$ to be a *almost-transparent enclave*, and show that it is possible to realise one-sided PSI between two parties running almost-transparent enclaves even if one of the parties is corrupted. Constructing a shell that realises the almost-transparent enclave can be achieved through a combination of the previous two shells, with the TransparentLeak additionally leaking the state of the work tape of the program before the command was executed, and the return value of all secure operation oracles. Leaking these values is required in the

There are some minor differences between our model and theirs: in their version of almost-transparent enclaves, once the initiatior issues a KEYEXCHANGE command, the receiving enclave is immediately notified and provided the symmetric key. Therefore, the initiator program needs to be run first (a natural constraint in their protocol). An additional difference from their model is our use of the idealised $\mathcal{F}_{\mathsf{crypto}}$ for all operations, rather than using a mix of ideal key exchange and concrete symmetric operations in their model. Therefore, we have to do an additional step to derive a symmetric key, rather than using the shared DH key directly.

The second relaxation, *semi-honest* enclaves, captures an adversarial manufacturer who is able to adaptively break into enclaves and extract historical transaction data. Note that in this setting, the party running the enclave does not need to be corrupted for the leakage to occur i.e. the adversary can cause leakage for any enclave run by a honest party. Despite the extreme vulnerability of this type of enclave to an adversarial manufacturer, it is still useful to construct some classes of private set intersection (distinct from the ones in the previous setting).

The shell for a Semi-honest enclave is defined as follows

```
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}}$ and $\mathbb{A} = \{\mathrm{CompleteLeak}\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
  $\mathsf{rec} \leftarrow []$
On message $(\text{RESUME}, \mathsf{input})$ from $G_{\mathsf{att}}^{mod}$:
  begin executing $\mathsf{input}$ on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  for next instruction $\mathsf{i}$ on virtual ITI do
    if $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ then
      allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
    else if $\mathsf{i} = (\textbf{return } v)$ then
      $\mathsf{rec} \leftarrow \mathsf{rec} \mid\mid (\mathsf{input}, args, \text{virtual ITI work tape})$
      return $v$ with source $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message COMPLETELEAK from $\mathcal{A}$:
  return $\mathsf{rec}$
```

The definition of the shell is quite simple, as it merely records the output of each resume execution and returns it to the adversary when it issues the CompleteLeak command. The message is sent directly to the shell rather than through a corrupted resume call to represent that it doesn't have to be issued by the calling party.

4.3.6 Shared Registry

We now give a shell to implement a single-writer multi-reader registry functionality for any subset of enclaves. The registry contains a linearisable list of values that any enclave in the set can read, but only one enclave can write into (in this case, the first enclave to complete a write). We give the adversary the ability to temporarily block or permanently censor corrupted parties, such that they can not access the registry for reading. If the number of censored replicas is greater than a certain quorum Q (a percentage of the registered parties) the registry is no longer able to guarantee termination of read/write operation, and will produce an error instead. If the writing enclave is censored, all subsequent write calls will fail but read calls from other enclaves can continue. The registry can be thought of as a shared single-writer ledger whose storage is distributed between enclaves, and is synchronised through a consensus mechanism; if less than Q of the total enclaves return a value, there are not enough live enclaves to establish consensus and thus the protocol terminates.

We define the following shell, where the adversarial oracle Censor_Q is parametrised by Q.

```
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}]$

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\mathrm{Read}, \mathrm{Write}\}$ and $\mathbb{A} = \{\mathrm{Block}, \mathrm{Censor}_\mathsf{Q}\}$
The extended identity of the shell is defined as $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message INSTALL from $G_{\mathsf{att}}^{mod}$:
    $j \leftarrow \perp$
    if virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ does not exist, create
    if ideal functionality $(\mathrm{RegCoord}[\mathrm{Q}], (\perp, \mathsf{idx}))$ does not exist, create
On message input from $G_{\mathsf{att}}^{mod}$:
    begin executing input on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    for next instruction i on virtual ITI do
        if $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ then
            allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
        else if $\mathsf{i} = \{\mathrm{Read}, (\mathrm{Write}, v)\}$ then
            if $j = \perp$ then
            send $Join$ to RegCoord[Q] on behalf of $(\text{prog}, (\text{eid}, \text{idx}))$; $j \leftarrow \top$
        send i to RegCoord[Q] through $(\text{prog}, (\text{eid}, \text{idx}))$ and receive v
        append $v$ to subroutine output tape of virtual ITI
    else if i $\in$(return $v$) then
        return $v$ with source $(\text{sh}_{\mathbb{O},\mathbb{A}}[\text{prog}], (\text{eid}||\text{pid}, \text{“att”}||\text{idx}))$
On message $(\text{CENSOR}, \varepsilon)$ from $G_{\text{att}}^{mod}$:
    send $(\text{CENSOR}, \text{pid})$ to RegCoord[Q]
```

Functionality RegCoord[Q]

State variables	Description
$P \leftarrow []$	List of enclaves participating in the registry
$C \leftarrow []$	List of censored enclaves
$V \leftarrow []$	List of registry values over time
$w \leftarrow \perp$	identity of writer enclave

```
On message JOIN from $(\text{prog}, (\text{eid}, \text{idx}))$:
    $P \leftarrow P \cup (\text{prog}, (\text{eid}, \text{idx}))$
    send $(\text{JOIN}, (\text{prog}, (\text{eid}, \text{idx})))$ to $\mathcal{A}$
On message $(\text{cmd}, v)$ from $(\text{prog}, (\text{eid}, \text{idx}))$:
    if eid is running on a corrupted party then
        query $\mathcal{A}$ with $(\text{READ}, (\text{prog}, (\text{eid}, \text{idx})))$ and receive the reply Block, $b$
    if $b \neq \top \wedge \frac{|C|}{|P|} < Q$ then
        if cmd=WRITE then
            if $w = \perp$ then $w \leftarrow P$
            if $w \neq P \vee P \in C$ then return Fail
            $V \leftarrow V \parallel v$
        send $(\text{CMD}, V, (\text{prog}, (\text{eid}, \text{idx})))$ to $\mathcal{A}$ return $V$
    elsereturn Fail
On message HEALTHCHECK from $(\text{prog}, (\text{eid}, \text{idx}))$:
    return $|P|, |C|$
On message $(\text{CENSOR}, (\text{prog}, (\text{eid}, \text{idx})))$ from $G_{\text{att}}^{mod}$:
    if eid is running on a corrupted enclave then
        $C \leftarrow C \parallel (\text{prog}, (\text{eid}, \text{idx}))$
        return
```

The above functionality allows any enclave shell to join the protocol as a registry party. The first shell who writes to the registry is locked in as w, the writer. Thereafter, only w can issue a new WRITE, which appends the value to the end of the registry, and all other registered parties receive the entirety of the registry on every READ[2] On any read and write, a corrupted party will query the adversary on whether they are allowed to proceed. The adversary can also permanently block an enclave by issuing a Censor message. If too many parties have been censored (i.e. the ratio between the number of censored parties and total registered parties is greater than Q), it is impossible for the registry to guarantee that the registry value is still safe, and the functionality fails.

We assume the functionality has access to some directory ITI that records whether enclaves are run by corrupted parties.

4.4 Relationships between G_{att}^{mod} variants

Having defined a variety of different G_{att}^{mod} functionalities with different sets $\mathbb{O}, \mathbb{A}$, we are now interested in exploring how they relate to each other. It is clear that all the shells described in the previous sections are a modelling tool, rather than a real implementation for that interface. As a downstream protocol designer, this level of abstraction is sufficient to detail the ideal behaviour of the oracles they require for their enclaves. To show that our model is realistic, however, we need to show that it is realisable in one way.

As we discussed in Section 2.2, there are a large number of TEE designs and enhancements that provide different features, as well as numerous attacks against real world implementations. Formalising what oracles are realised by a specific TEE implementation is a non-trivial task, and once a set is finalised, the discovery of new attack oracles might invalidate the security of any proofs using it as a hybrid (or at least making the protocol vulnerable "in the real world"). Rather than taking this bottom-up approach, we propose to go the other direction: showing that strong TEE setups, which we know are not possible to realise with our current implementations (despite their usage in security proofs), can be gradually realised through a less powerful abstraction.

Our intuition is that, given two versions of G_{att}^{mod} which sign over the same measurement functions, a "weaker" setup (G_{att}) that has either fewer features or more attacks

[2]The functionality could be made more efficient by keeping track of what values have been read by each group member, and only downloading the difference on a read.

can UC-emulate the stronger one (G'_{att}). If there is a "wrapper" protocol around G_{att} that for all enclave programs running on it can emulate the missing feature oracle, or mitigate the additional adversarial oracle, the combination of the wrapper protocol with the G_{att} setup is at least as strong as G'_{att}.

This section sketches how to design such a protocol to show the UC-emulation between any two $G_{\mathsf{att}}, G'_{\mathsf{att}}$ setups. Our treatment aims to be generic and provide a universal compiler protocol, but we are aware that our design will not work for all combinations of oracles running arbitrary programs. Our protocols and proofs should be seen as templates to be adapted to the specific setups under consideration.

4.4.1 Adding a Feature oracle

To fully capture the modular power of our new formalisation, we show how to add a new feature to a TEE instance, increasing the size of its feature oracle set. We want to show that a TEE that has native access to that feature (through an oracle) is indistinguishable from one that does not and has to implement it through runtime code. Depending on its complexity, a feature can be implemented by just running some additional computation within the enclave itself, by calling out to a library running within an assisting enclave on the same party, or by conducting an interactive protocol with multiple remote parties. We can represent these type of runtime behaviour as a UC protocol that provides the same interface and guarantees of the missing feature oracle.

More formally, we consider two instantiations of attested execution G_{att} and G'_{att} (both modular), with feature oracles $\mathbb{O}, \mathbb{O}'$, respectively, where $\mathbb{O} \subset \mathbb{O}'$. Let $I = \mathbb{O}' \setminus \mathbb{O}$. The adversarial oracles $\mathbb{A}$ and attestation signature function $\mathbb{S}$ are shared between G_{att} and G'_{att}. We now define a new "wrapper" protocol $\mathcal{W}$ which uses G_{att} as a subroutine and UC-realises G'_{att} by implementing the interface for I in the real world.

$\mathcal{W}$ takes the same parameters as G_{att}^{mod}, and in addition the two functions $\mathsf{map}^{\mathsf{L}}, \mathsf{map}^{\mathsf{R}}$, and the code of enclave program $\mathrm{W}^{I}[\cdot]$. Function $\mathsf{map}^{\mathsf{R}}$ takes the set of G_{att}^{mod}-enabled parties, and chooses a subset to run assisting enclaves that any party can rely on (the parties chosen by $\mathsf{map}^{\mathsf{R}}$ do not have to be honest). $\mathsf{map}^{\mathsf{L}}$ returns a set of local assisting enclave programs the party should install locally, and a next message function for $\mathrm{W}^{I}[\cdot]$.

$\mathrm{W}^{I}[\mathsf{prog}, \mathsf{nextmsg}]$ is a "wrapper" enclave that instruments prog with additional code such that, when prog attempts to use interface I, the next message function begins executing I as a protocol. Function nextmsg observes the current state of the enclave, and chooses the command required to start the I-protocol execution. The command issued by nextmsg will either be run as a local subroutine in the enclave wrapper code itself, by another enclave installed locally (as instructed by $\mathsf{map}^{\mathsf{L}}$), or by a remote party (in an enclave created through $\mathsf{map}^{\mathsf{R}}$). nextmsg is aware of the details of each assisting enclave, such as their enclave ID or what party they are installed on.

If the next command issued by nextmsg is received by the assisting enclave it is destined for (a corrupted party could diverge from the protocol and choose not to deliver the message), it executes the requested subroutine, produces its own next command, and forwards it to the party that should execute it. Eventually, the original $\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$ will receive a final message, and return the result value of I to the prog oracle call. Essentially, the program that implements I is compiled into a multi-party computation between the enclaves. We do not require a full-fledged secure MPC protocol to execute I, however, due to the integrity guarantees of attestation, as the only possible malicious behaviour of a participant is dropping messages (known as the omission corruption adversarial model in MPC [64]). Within the execution of the next message functions, enclaves are able to construct an authenticated or secure channel through attestation. We do not give a description of how this is done, and refer the reader back to the construction of the secure channel in the Steel protocol in Section 3.3.

The $\mathcal{W}$ protocol (Figure 4.4) proceeds as follows. During initialisation, it calls the $\mathsf{map}^{\mathsf{R}}$ function to produce a list of supporting enclaves E_i^R run by a subset of reg parties, initialises G_{att}^{mod} with the appropriate parameters, and requests each selected party to install the wrapped E_i^R. It then returns a public list of all assisting parties and the associated enclave IDs.

On a call from pid_i to install some program prog_i, if prog_i does not include any call to I, it installs a wrapped version with dummy next message function ε. Otherwise, it runs $\mathsf{map}^{\mathsf{L}}()$ to produce a list of local assisting enclave programs to be installed by the same party, and a next message function $\mathrm{nextmsg}_{\emptyset}$. The party installs all such enclaves, runs their initialisation subroutine, and creates a new message function nextmsg that is a wrapper around $\mathrm{nextmsg}_{\emptyset}$ aware of the assisting parties enclave IDs. $\mathsf{map}^{\mathsf{L}}$ makes nextmsg and all E_i^L programs aware of the enclave IDs for any E^R parties, and assists $\mathrm{W}^{I}[\cdot]$ in generating the appropriate next commands to implement I along with the assisting protocols.

On a resume call from its local party pid_i to execute command cmd on arguments

```
Protocol $\mathcal{W}[\lambda, G_{\mathsf{att}}^{mod}, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}, \mathsf{map}^{\mathsf{R}}, \mathsf{map}^{\mathsf{L}}, \mathrm{W}^{I}[\cdot]]$

On message INITIALISE from a party P:
  $[(\mathrm{E}_1^R, \mathsf{pid}_1), \ldots, (\mathrm{E}_n^R, \mathsf{pid}_m)] \leftarrow \mathsf{map}^{\mathsf{R}}(\mathsf{reg})$
  send INITIALISE to $G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}]$
  let $\bar{\mathrm{E}}^R \leftarrow []$
  for $i \in \{0, \ldots, n\}$ do
    send $(\text{INSTALLREQUEST}, \mathrm{E}_i^R)$ to $\mathsf{pid}_i$ and receive $\mathsf{eid_i}$
    $\bar{\mathrm{E}}^R \leftarrow \bar{\mathrm{E}}^R || \mathsf{pid}_i, \mathsf{eid_i}$
  return $\bar{\mathrm{E}}^R$
On message GETPK from a party P:
  send GETPK to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{vk}$
  return $\mathsf{vk}$
On message $(\text{VERIFY}, \sigma, m)$ from a party P:
  if $m$ is an attestation measurement that contains a commitment to some program with
  code $\mathrm{E}_i^R$ or $\mathrm{E}_i^L$ then
    return $\perp$
  else
    send $(\text{VERIFY}, \sigma, m)$ to $G_{\mathsf{att}}^{mod}$ and receive $v$ and return v
On message $(\text{INSTALL}, \mathsf{prog})$ from a party P where $P.\mathsf{pid} \in \mathsf{reg}$:
  if $I \in \mathsf{prog}$ then
    $(\mathrm{nextmsg}_{\emptyset}, (\mathrm{E}_1^L, \ldots, \mathrm{E}_n^L)) \leftarrow \mathsf{map}^{\mathsf{L}}(\bar{\mathrm{E}}^R)$
    let $\bar{\mathrm{E}}^L \leftarrow []$
    for $i \in \{1, \ldots, n\}$ do
      send $(\text{INSTALL}, \mathrm{E}_i^L)$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{eid_i}$
      send $(\text{RESUME}, \mathsf{eid_i}, \text{INIT})$ to $G_{\mathsf{att}}$
      $\bar{\mathrm{E}}^L \leftarrow \bar{\mathrm{E}}^L || \mathsf{eid_i}$
    let $\mathrm{nextmsg}(x) \leftarrow \mathrm{nextmsg}_{\emptyset}(x, \bar{\mathrm{E}}^L)$
    send $(\text{INSTALL}, \mathrm{W}^I[\mathsf{prog}, \mathsf{nextmsg}])$ to $G_{\mathsf{att}}$ and receive $\mathsf{eid_{prog}}$
    send $(\text{RESUME}, \mathsf{eid_{prog}}, \text{INIT})$ to $G_{\mathsf{att}}$
  else
    send $(\text{INSTALL}, \mathsf{sid}, \mathrm{W}^I[\mathsf{prog}, \varepsilon])$ to $G_{\mathsf{att}}$ and receive $\mathsf{eid_{prog}}$
  return $\mathsf{eid_{prog}}$
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ from a party P with $\mathsf{pid}_i$:
  send $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{out}, \sigma$
  while $\mathsf{out} = (\text{RESUMEREQUEST}, \mathsf{pid}, \mathsf{eid}', v)$ do
    if $\mathsf{pid} = \mathsf{pid}_i$ then
      $(\mathsf{out}, \sigma') \leftarrow \text{RESUME}(\mathsf{eid}', (v, \sigma, \top)))$
    else
      send $(\text{RESUMEREQUEST}, \mathsf{eid}', (v, \sigma))$ to pid and await
      if next message $m, \sigma'$ on input tape does not start with RESUMEREQUEST then
  ignore
      else $\mathsf{out} \leftarrow m, s\sigma'$
  return $\mathsf{out}, s$
On message $(\text{INSTALLREQUEST}, \mathsf{prog})$ from a party $P \in \mathsf{reg}$:
  send $(\text{INSTALL}, \mathsf{prog})$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{eid}$
  return $\mathsf{eid}$
On message $(\text{RESUMEREQUEST}, \mathsf{eid}, \mathsf{input})$ from a party $P \in \mathsf{reg}$:
  send $(\text{RESUME}, \mathsf{eid}, \mathsf{input}))$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{output}, \sigma$
  return $(\mathsf{output}, \sigma)$
```

Figure 4.4: The wrapper protocol

args for enclave $\mathrm{W}^I[\mathsf{prog_i}, \mathrm{nextmsg}]$, the enclave wrapper (described in Figure 4.5) begins executing the code of $\mathsf{prog_i}$ with those inputs. Once the program makes a subroutine call to I, the wrapper stops the internal program execution and calls the nextmsg function, which returns the PID, Enclave ID and some command that needs to be executed to begin computing the value for the subroutine call. The enclave returns these to its local party with the special keyword RESUMEREQUEST, and waits for a next activation. When the party receives this return value, it knows that cmd(*args*) did not terminate. Instead, it passes the RESUMEREQUEST and associated command on to the appropriate party, or, if the destination PID is pid_i, activates one of its local enclaves, including $\mathrm{W}^I[\mathsf{prog_i}, \mathrm{nextmsg}]$ itself. When resuming an enclave as part of the I computation, the local party can set input flag $\top$ as part of the RESUME arguments to indicate that the command being executed is not part of the normal $\mathsf{prog_i}$ code. pid_i waits to receive the next message, and once again passes it on to one of its local enclaves, and forwards the resulting RESUMEREQUEST. Eventually, when the PID and EID returned by nextmsg are $\perp$, the computation of I has terminated, and the wrapper can pass back v as its return value to the internal execution of $\mathsf{prog_i}$. Whenever the enclave returns with an intermediate message, the latest attestation signature should always be bundled with the next message input for the receiver party. Attestation validation logic is defined in the code of $\mathcal{W}$ for all appropriate messages, and interacts directly with the G_{att}^{mod} verification request through a call to the AttestVerify protocol. When a user requests verification of one of these intermediate attestation signatures from outside one of the participating enclaves, the protocol always returns $\perp$.

It is convenient for our purposes to model the code of $\mathrm{W}^I[\cdot]$ using a UC shell, since its behaviour is similar to some of the shells we constructed in the previous section. The two types of shell are complementary: UC structured protocols support nesting shells, so we instantiate the $\mathrm{W}^I[\cdot]$ as a subroutine of $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\cdot]$. Formally, the program installed by $\mathcal{W}$ is $\mathrm{W}^I[p, f]$, but using the shell means that we don't have to define its full source code (or more likely, a compiler program that interleaves calls to next function f throughout p). Additionally, the oracle sets provided by the combined interface $\mathbb{O} \parallel I$ is intuitively equivalent to the oracle sets of G'_{att} (see Figure 4.6 for a graphical representation).

We now provide the following conjecture:

Conjecture 4.1. *Let* $G_{\mathsf{att}} = G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}], G'_{\mathsf{att}} = G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}', \mathbb{A}, \mathbb{S}], \mathbb{O}' \setminus \mathbb{O} = I$. *For any enclave wrapper* $\mathrm{W}^I[\cdot]$ *which, combined with functions* $\mathsf{map}^\mathsf{L}, \mathsf{map}^\mathsf{R}$ *implements the difference between the shells* $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\cdot]$ *and* $\mathrm{sh}_{\mathbb{O}',\mathbb{A}}[\cdot]$*, it is possible to show*

```
Shell $\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$ (Template)

The identity of the shell is $(\mathsf{eid}, \mathsf{idx})$
The parent shell extended identity is $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathrm{W}^{\mathrm{I}}[\mathsf{prog}]], (\mathsf{eid}||\mathsf{pid}, \text{"att"}||\mathsf{idx}))$

On message $(\mathrm{cmd}, args, r)$ from $(\mathsf{eid}||\mathsf{pid}, \text{"att"}||\mathsf{idx})$:
  if virtual ITI $(\mathsf{prog}, (\mathsf{eid}||\text{"wrapped"}, \mathsf{idx}))$ does not exist then create
  if $r = \perp$ then
      let $\mathsf{input} \leftarrow (\mathrm{cmd}, args)$
      begin executing input on $(\mathsf{prog}, (\mathsf{eid}||\text{"wrapped"}, \mathsf{idx}))$
      for next instruction $\mathsf{i}$ on virtual ITI do
          if $\mathsf{i} \notin I$ then
              // Execution of i is delegated to the higher order shell
              allow $(\mathsf{prog}, (\mathsf{eid}||\text{"wrapped"}, \mathsf{idx}))$ to execute $\mathsf{i}$
          else
              $(\mathsf{pid_j}, \mathsf{eid_j}, v) \leftarrow$ nextmsg(tapes of virtual ITI)
              while $(\mathsf{pid}_j, \mathsf{eid_j}) \neq (\perp, \perp)$ do
                  send $(\textsc{resumeRequest}, (\mathsf{pid_j}, \mathsf{eid_j}, v))$ to $(\mathsf{eid}||\mathsf{pid}, \text{"att"}||\mathsf{idx})$ and await
                  if next message on the input tape is $(\mathrm{cmd}', args', \top)$ then
                      execute $(\mathsf{pid_j}, \mathsf{eid_j}, v) \leftarrow \mathrm{cmd}'(args', \top)$
                  else
                      ignore
              append $v$ to subroutine output tape for $(\mathsf{prog}, (\mathsf{eid}||\text{"wrapped"}, \mathsf{idx}))$
              // The loop terminates when nextmsg() returns $(\perp, \perp, v)$
  else
      // $r = \top$ as the result of an $I$ computation, execute code in subroutine cmd
      execute $\mathrm{cmd}(args)$
      return nextmsg(tapes of virtual ITI)
```

Figure 4.5: Template for the internal wrapper shell. A complete definition of the shell requires an implementation for any additional CMD that might be requested by the next message functions

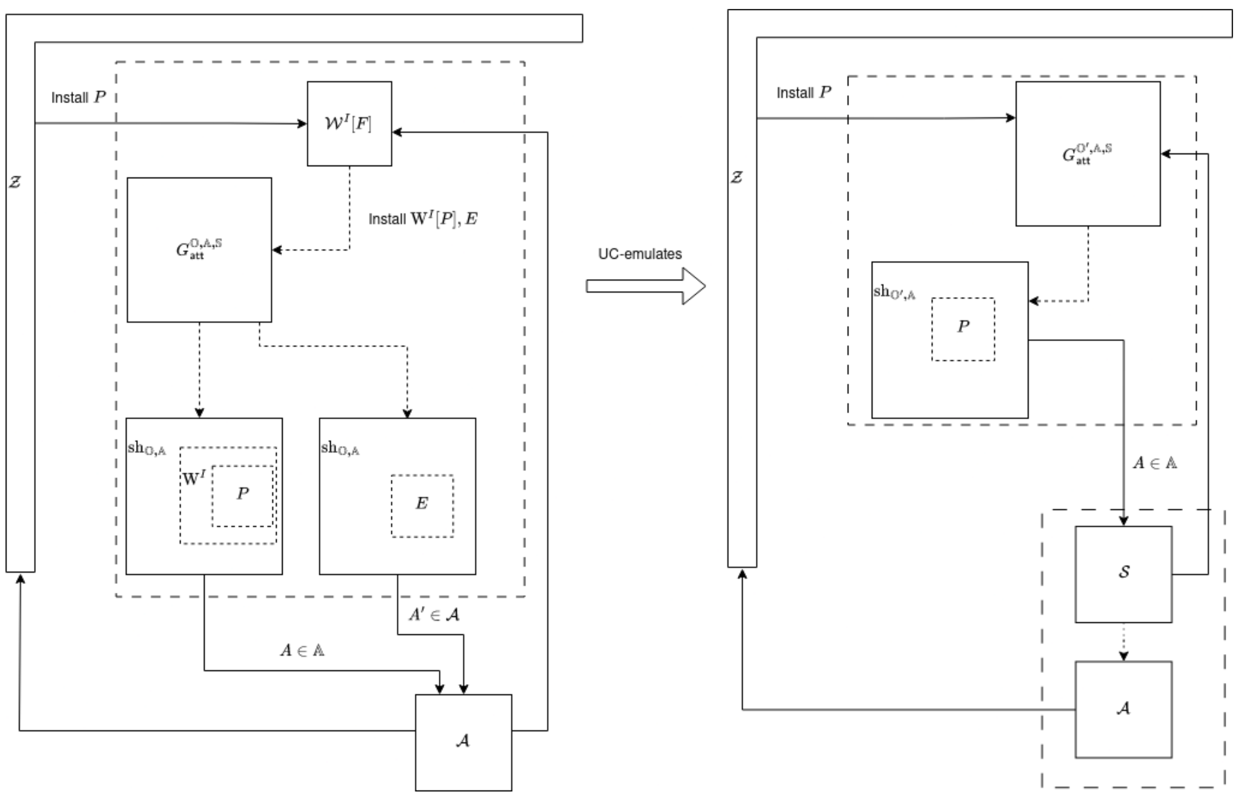
Install P
$\mathcal{W}^I[F]$
$\mathcal{Z}$
Install $\mathrm{W}^I[P], E$
$G_{\text{att}}^{\mathbb{O},\mathbb{A},\mathbb{S}}$
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}$
W^I
P
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}$
E
$A' \in \mathcal{A}$
$A \in \mathbb{A}$
$\mathcal{A}$
UC-emulates
Install P
$\mathcal{Z}$
$G_{\text{att}}^{\mathbb{O}',\mathbb{A},\mathbb{S}}$
$\mathrm{sh}_{\mathbb{O}',\mathbb{A}}$
P
$A \in \mathbb{A}$
$\mathcal{S}$
$\mathcal{A}$

Figure 4.6: Protocol $\mathcal{W}$ can add a shell to $G_{\text{att}}^{\mathbb{O},\mathbb{A}\mathbb{S}}$ enclaves to UC-emulate the missing feature oracles I from $G_{\text{att}}^{\mathbb{O}',\mathbb{A}\mathbb{S}}$

that protocol $\mathcal{W}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}, \mathsf{map}^{\mathsf{R}}, \mathsf{map}^{\mathsf{L}}, \mathrm{W}^I[\cdot]]$*, which instantiates* G_{att} *as a subroutine, UC-emulates* G'_{att}

Without a precise definition of protocol $\mathcal{W}$ and the interface it is implementing, or the preexisting interfaces for $\mathbb{O}$ and $\mathbb{A}$, it is difficult to provide evidence that $\mathcal{W}$ in the presence of G_{att} UC-emulates G'_{att}. We now provide some guidelines on how a simulator for such theorems of special instances of this conjecture might be structured; however, depending on the nature of the programs installed, the wrapper code, or the shared oracles between the two setups, a different simulation strategy might be needed. For example, if the adversary is able to directly observe the source code of an enclave while it is executing, the simulation will not work. It might be possible for some of this cases where the below simulation strategy does not work to add some backdoor code in the $\mathrm{W}^I[\cdot]$ description to give the simulator some additional powers (see the Steel proof in Section 3.4 or Pass, Shi, and Tramèr [230]).

We describe simulation for the three possible protocol topologies implementing I:

1. We begin our simulation sketch for the case of a wrapper protocol $\mathcal{W}$ where neither $\mathsf{map}^{\mathsf{R}}$ or $\mathsf{map}^{\mathsf{L}}$ functions returns any additional enclave i.e. the wrap-

per $\mathrm{W}^{I}[\cdot]$ can implement I without relying on any external assistance. During the global functionality initalisation phase, the simulator observes the signature algorithm s chosen by the environment through the dummy adversary, and provides G'_{att} with a new algorithm s' which applies s over the transformation $F(\mathsf{meas})$. F takes a measurement string that contains an identifier for program prog, and replaces it with an identifier for $\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$ (for an appropriate value of nextmsg), as discussed in Section 4.2. Attestations produced by an enclave prog in the G'_{att}-hybrid world are thus indistinguishable from those produced by the equivalent wrapped enclave in the G_{att}-hybrid world. Therefore, the simulator can simply block any installations of an un-wrapped program that requires access to I with MissingInstructionError, and replace installations of wrapped programs with the unwrapped version on the G'_{att} functionality. Honest parties in protocol $\mathcal{W}$ do not install any unwrapped program, and no external session will have direct access to G_{att} since it is installed as a $\mathcal{W}$ subroutine. If the (local) adversary attempts to install an unwrapped program to G_{att} directly, the simulator can run the program "in its head" without going through G'_{att}, and use the algorithm s provided by the dummy adversary for the environment for producing plausible attestation signatures for the unwrapped code. The signatures will not verify through any calls to the ideal verification subroutine, as they wouldn't for honest parties of $\mathcal{W}$, but they will look legitimate to environment through running the local verification algorithm that corresponds to s.

2. When $\mathsf{map}^{\mathsf{R}}$ does not install any assisting enclaves, but $\mathsf{map}^{\mathsf{L}}$ does, the simulator instantiates the same signature scheme as in the previous case (by adding the F transformation).

   When it receives a request to install any enclave with code E_i^L, it generates a plausible enclave ID and returns it, without actually installing the enclave in G_{att}^{mod}. While we do not explicitly define an enclave ID generation algorithm for G_{att}^{mod}, we assume that the probability of sampling the same ID is negligible. The simulator then ensures that, before a corrupted party requests to install some enclave $\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$, it has requested to install all necessary E_i^L enclaves produced by $\mathsf{map}^{\mathsf{L}}()$, and has given a value of nextmsg with the appropriate enclave IDs, otherwise $\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$ would not be able to verify them for attestation.

   Whenever the adversary resumes the program enclave, the simulator runs the input in its head to determine whether it contains any calls to I. If it does, it calls the next message function nextmsg on the partial result, and uses the signing key generated during initialisation by the adversary to produce signing algorithm s, and uses it to sign a RESUMEREQUEST message. If the adversary then tries to resume the receiver enclave, the simulator executes the related command in its head and returns the next RESUMEREQUEST message. Any attempts from the adversary to verify one of the intermediate attestation messages directly is dropped, since the protocol does not let parties verify these attestations either (they are however likely to be verified by the code of the wrapper enclave as part of its next command execution). Once it is satisfied that the adversary has provided the appropriate sequence of messages to fully compute I, it sends the initial original input to the unwrapped program in G'_{att}. If the feature shell triggers any adversarial interaction, it uses the values provided by the adversary through RESUMEREQUEST messages to maintain a consistent state with the $\mathcal{W}$ interactions. Any interactions with the adversary through attacks or feature requests unrelated to I are captured by the ideal shell run by G_{att}, so no additional simulation is required for them.

3. Finally, in the case of the $\mathsf{map}^{\mathsf{R}}$ function requesting multiple enclaves across a variety of parties, the simulator initialises G^{mod}_{att} with the same signature algorithm as before. It then calls the $\mathsf{map}^{\mathsf{R}}$ function and sends the resulting resume requests to corrupted parties, but installs the assisting enclaves for honest parties on a machine it controls, and produces the appropriate list of assisting enclave IDs, $\bar{\mathrm{E}}^{R}$.

   Like in the previous case, on an enclave installation request, it installs a non-wrapped copy of any enclaves requested by corrupted parties, as long as they have installed all the related local assisting enclaves. Simulation proceeds as in the previous case, except that the simulator also ensures that any remote RESUMEREQUEST message is delivered (i.e. the appropriate messages on the network are not censored). When a next command is sent to a remote assisting party run by some honest user, the simulator does not pass it on, and runs the command on its local copy to find out the next message location, using its copy of the s algorithm to sign plausible attestations (including faking the party ID if using non-anonymous attestation) Finally, if the computation succeeds, it calls the unwrapped enclave in G'_{att} as before. Any attempts by a corrupted party to send a RESUMEREQUEST to honest enclaves outside of the correct sequence of events is dropped.

General replacement of global setups As we discussed in Section 2.1.2.2, it is not possible to prove, in the general case, that a protocol UC-emulates a global subroutine. A well formed replacement statement needs to account for the context emulation statement the global subroutine is being invoked in.

Intuitively, since the adversarial oracle sets for the $G_{\mathsf{att}}, G'_{\mathsf{att}}$ functionalities considered are the same, replacing the global functionality G'_{att} with a G_{att}-hybrid protocol $\mathcal{W}$ to provide the missing feature interface I should generally be safe, as a higher level simulator that interacts with TEEs as part of a protocol subroutine will have the same interface for attacks. However, given the general nature of our conjecture, we can not conclusively say that the implementation of I provided by $\mathcal{W}$ communicates with the adversary in the same manner as the ideal implementation of I provided by the G'_{att} shell. Indeed, the role of the $\mathcal{W}$ to G'_{att} simulator is to reconciling any such difference. We therefore have to analyse two distinctive cases.

Theorem 4.1. *Let $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ be any G_{att}^{mod} setups and a wrapper protocol such that Conjecture 4.1 holds, and additionally G'_{att} UC-emulates $\mathcal{W}$. For any protocol ρ in the presence of G'_{att} that UC-emulates some $\mathcal{F}$ in the presence of G'_{att}, ρ in the presence of $\mathcal{W}$ UC-emulates $\mathcal{F}$ in the presence of $\mathcal{W}$.*

The statement follows from the composition theorem of [33, Theorem 3.3]. Showing that G'_{att} UC-emulates $\mathcal{W}$ (i.e. in conjunction with 4.1, $\mathcal{W}$ and G'_{att} are UC-equivalent) involves constructing a new simulator $\mathcal{S}'$ such that $\mathsf{EXEC}_{G'_{\mathsf{att}},\mathcal{A},\mathcal{Z}} \approx \mathsf{EXEC}_{\mathcal{W},\mathcal{S}',\mathcal{Z}}$.

During the setup phase, $\mathcal{S}'$ instantiates G_{att}^{mod} with the inverse transformation for attestation signatures described in the proof of 4.1 i.e. for any attestation measurement that includes an identifier for some program with code $\mathrm{W}^{I}[\mathsf{prog}, \cdot]$ and replaces it with an identifier for prog. Thereafter, the behaviour of $\mathcal{S}'$ consists of simply forwarding any input from the environment to the protocol $\mathcal{W}$ (including allowable attacks in $\mathbb{A}$), and after a RESUME, execute any associated RESUMEREQUEST for corrupted parties without modifying their inputs or showing the result to the environment, except for any adversarial leakage consistent with what would be produced by the shell implementation for G'_{att}. When $\mathcal{W}$ returns the output of the RESUME and associated attestation message, $\mathcal{S}'$ only forwards this result and its attestation (with the wrapper code removed by the F^{-1}

If the shell implementing feature I in the G'_{att} world includes direct communication with the adversary that is not fully equivalent by the messages produced by the supporting enclaves in $\mathcal{W}$, the simulation will fail. For such protocols we need to consider a weaker setting, where we fix the feature simulator within the ideal subroutine available to the higher level protocols.

Theorem 4.2. *Let $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ be any G^{mod}_{att} setups and a wrapper protocol such that Conjecture 4.1 holds for some simulator S. Let G^{S}_{att} be the combination of G'_{att} and S; for any protocol ρ in the presence of G^{S}_{att} that UC-emulates some $\mathcal{F}$ in the presence of G^{S}_{att}, ρ in the presence of $\mathcal{W}$ UC-emulates $\mathcal{F}$ in the presence of $\mathcal{W}$.*

The statement above directly follows from [85, Lemma 1].

4.4.2 Removing Adversarial Interfaces

Just like the above protocol allows increasing the size of a TEE feature oracle interface set, we now formulate a corresponding protocol to reduce an enclave's attack surface. For many types of enclave attacks, there are cryptographic or distributed protocols that can provide some degree of protection. We can use these protocols to construct a new functionality with a smaller adversarial interface set. A core difference from the oracle interface implementation of the previous section, however, is that, rather than interrupting the execution of a normal enclave program for a specific instruction to run a protocol between supplementary enclaves, it is necessary to run the defensive protocol from the start of the execution. Since the adversary could mount the attack during or between arbitrary resume operations, the protocol might need to execute certain instructions before or independently from an attack, such as establishing a secure channel with an assisting enclave.

As in the previous section, given two (modular) implementations of attested executions $G_{\mathsf{att}}, G'_{\mathsf{att}}$ with adversarial interfaces $\mathbb{A}, \mathbb{A}'$ respectively, and shared $\mathbb{O}$ and $\mathbb{S}$, we define a wrapper protocol $\mathcal{W}$ (for non-empty $A = \mathbb{A} \setminus \mathbb{A}'$) that uses G_{att} as a subroutine and UC-realises G'_{att}.

Protocol $\mathcal{W}$ is defined in the same way as $\mathcal{W}$. As we remarked above, the only difference between the two protocols is that $\mathrm{W}^{A}[\mathsf{prog}, \mathsf{nextmsg}]$ never executes the internal protocol prog directly. Instead, the nextmsg() function now takes the code prog as an additional argument, and compiles it into the code for local enclaves $\mathrm{E}^{L}_{i\in\{1,\ldots,n\}}$. When the party installs all enclaves $\mathrm{W}^{A}[\mathsf{prog}, \mathsf{nextmsg}], \mathrm{E}^{L}_{1} \ldots, \mathrm{E}^{R}_{n}\}$, it immediately resumes them with INIT, which allows the enclaves to conduce any necessary setup operations. Thereafter, on a resume call to prog, $\mathrm{W}^A[\mathsf{prog}, \mathsf{nextmsg}]$ always begins its execution by running nextmsg first. When nextmsg returns $(\bot, \bot, v)$, this indicates that the resume call has completed, and the enclave returns v to the party.

The protocol $\mathcal{W}$ only protects against the attacks in A; all other attacks in $\mathbb{A}'$ are still allowable in both worlds.

We omit the formal description or the protocol or wrapper enclave due to their similarity to the one in the previous section. Likewise, we omit a further summary of the simulation techniques for showing that $\mathcal{W}$ UC-emulates G'_{att}, in favour of adopting a concrete example in Section 4.5. However, we do state the following for completeness:

Conjecture 4.2. *Let* $G_{\mathsf{att}} = G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}], G'_{\mathsf{att}} = [\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}', \mathbb{S}], \mathbb{A} \setminus \mathbb{A}' = A$. *For any enclave wrapper* $\mathrm{W}^A[\cdot]$ *which, combined with functions* $\mathsf{map}^\mathsf{L}, \mathsf{map}^\mathsf{R}$ *implements the difference between the shells* $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\cdot], \mathrm{sh}_{\mathbb{O},\mathbb{A}'}[\cdot]$*, it is possible to show that protocol* $\mathcal{W}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}, \mathsf{map}^\mathsf{R}, \mathsf{map}^\mathsf{L}, \mathrm{W}^A[\cdot]]$*, which instantiates* G_{att} *as a subroutine, UC-emulates* G'_{att}

Theorem 4.3. *Let* $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ *be any* G_{att}^{mod} *setups and a wrapper protocol such that Conjecture 4.1 holds. For any protocol* ρ *in the presence of* G'_{att} *that UC-emulates* $\mathcal{F}$ *in the presence of* G'_{att}*,* ρ *in the presence of* $\mathcal{W}$ *UC-emulates* $\mathcal{F}$ *in the presence of* $\mathcal{W}$*.*

We claim the latter theorem holds because the adversarial interface is smaller in the ideal world, so there is no additional attack that was used by the ρ to $\mathcal{F}$ simulator which is no longer available with the introduction of the protocol. This is the inverse scenario of which Badertscher, Hesse, and Zikas [33] are concerned, where the real world global protocol includes fewer attacks that the ideal world global functionality. Therefore, the theorem holds due to the composition theorem of [33, Theorem 3.10], as the ρ to F simulator is $\mathcal{W} \setminus \mathbb{A}'$-agnostic (i.e. the simulator does not interact with $\mathcal{W}$ except for using adversarial interfaces in $\mathbb{A}'$ - that is, everything except for A). This is true because A is not a valid adversarial interface in G'_{att}. Therefore, if simulator of the pre-condition is able to simulate the protocol without using A, the same simulator will equally apply to the statement where G'_{att} has been replaced with $\mathcal{W}$.

4.4.3 Interactions Between Features and Attacks

When defining the transformation between two versions of G_{att}, it is important to think carefully about specifying the necessary requirements. Just like the defence protocol to remove some adversarial attack might require specific feature oracles (therefore the addition of attack A requires a lower bound for $\mathbb{O}$), there will be classes of attacks that can break security of most protocols without access to some adequate feature oracles to construct a protection mechanism (setting an upper bound to what attacks can be introduced in $\mathbb{A}$).

Additionally, in some cases the addition of a new feature will also imply the expansion of adversarial attacks. Consider the addition of explicit storage and fetching capabilities described in Section 4.3.3. By adding those external oracle calls, we are also forced to provide an adversarial oracle to abort the program. While it would be possible to consider a version of $G_{\mathtt{att}}^{mod}$ where only the new interfaces were added, it would be hard to justify as the natural implementation of that feature requires handing off control of the memory to untrusted permanent storage. Of course, a novel TEE architecture could allow a more secure way to implement storage and fetching without exposing the enclaves to adversarial crashes. Our goal for $G_{\mathtt{att}}^{mod}$ is not to be prescriptive with what kind of (ideal) TEE objects should be used as assumptions in cryptographic protocols; however we recommend caution when designing a new variant of $G_{\mathtt{att}}^{mod}$ with complex or unrealistic features.

Another illustrative example could be the introduction of cloning [175]. This feature allows efficient enclave creation, as it instantiates a second copy of an enclave including its memory (equivalent to normal process forking in operating systems). Depending on the implementation, the addition of this feature might however give the adversary additional power, as it could now be able to swap memory regions for each of the two versions of the enclave interactively, effectively executing a forking attack not tied to rollback (where the remote party is not able to distinguish which of the two enclaves it is communicating with, and the adversary can interactively swap and censor messages between the two). While this specific attack can be easily mitigated with another wrapper protocol that augments sealing with freshness values, it will require an additional explicit transformation and corresponding level of shells.

Our theorems in this section only show a single step $G_{\mathtt{att}}^{mod}$ oracle change (through feature addition and attack removal). Unlike the oracle shells in Section 4.3, which have to be manually integrated to provide the appropriate functionality for the set of oracles chosen (although in many cases the shell changes are trivial), it should be easy for some oracle combinations to repeatedly apply Conjectures 4.1 and 4.2 without modifying the protocols.

We note that in some cases the oracle transformation protocols given above might not be simulatable for all possible enclave programs. In those cases, it is still possible for a program designed to run in G'_{att} to run in the G_{att}-hybrid world where the oracle feature is not available, or be secure even if G_{att} allows an attack not in G'_{att}. Such substitution require to be proven on a case by case basis, but the observation is consistent with the state of the art of TEE program design, where mitigations for certain attacks exist only if the program is "well-written" (e.g. memory safe or using oblivious primitives) or does not use certain functions (see [251, Table 1]).

4.5 Implementing Rollback protection from Registers

We now give an example of one the equivalences described in the previous section, with the aim of addressing the rollback attacks described at the start of the chapter. Our construction relies on the well-known observation in the literature that a monotonic counter or a trusted storage services can be used to prevent rollback attacks (see Section 2.2.3.1). Although the protocol equally applies to the related class of forking attacks, we do not explicitly address them in this section for simplicity.

To construct the protocol, we require our target enclave to support the trusted Store, Fetch interfaces we described in Section 4.3.3, as well as an oracle Meas, which returns a digest (such as a hash) over the state of the enclave's virtual ITI. We construct a simple protocol $\mathcal{W}$ as described in Section 4.4, that removes the Rollback interface from an ideal G^{mod}_{att} where $\mathbb{A}$ includes Abort.

The intuition for the protocol is that the shell can store the digest of the latest copy of the internal enclave measurement in persistent storage at the end of every RESUME. When enclave execution starts, the shell can fetch the stored measurement digest and compare it with the measurement for the current state as returned by Meas. If the two states match, the enclave can be safely executed; otherwise, the state must have been tampered with, and the function aborts. We denote this sequence of operations as wrapper-subroutine MEASEXEC. If every resume operation uses MEASEXEC, the adversary is not able to execute a rollback attack, but will effectively abort the enclave. Defining a rollback protection protocol by relying on the usage of safe memory might seem like a circular definition - if the enclave has access to trustworthy Store, Fetch oracles, why not just store the entirety of memory using this interface? We discuss more realistic protection mechanisms in Chapter 6 as material for future work, but we believe that the current setting is still valuable, as it minimises usage of the size of data stored in the trusted memory (as well as providing an easy to explain protocol to prove an instance of Conjecture 4.2).

To provide the formal definition for the protocol, we define functions $\mathsf{map}^{\mathsf{R}}()$ and $\mathsf{map}^{\mathsf{L}}()$ that produce no supporting enclaves. Function $\mathsf{map}^{\mathsf{L}}()$ defines a next message function $\mathrm{nextmsg}_{\mathrm{meas}}()$, which determines how to execute the wrapped program. When the enclave state is at the beginning of executing a RESUME instruction, $\mathrm{nextmsg}_{\mathrm{meas}}$ runs the MEASEXEC subroutine of $\mathrm{W}^{A}[\cdot]$. Subroutine MEASEXEC checks that the current measurement of the enclave's state corresponds to the last state saved in storage, before executing the input subroutine, and updating the storage with the resulting new state. If MEASEXEC aborts, $\mathrm{nextmsg}_{\mathrm{meas}}$ returns $(\bot, \bot, \text{“abort”})$, while if it terminates successfully with value v, it returns $\bot, \bot, v$; in both cases, the enclave returns the values to its caller.

The code of the shell that implements the $\mathrm{W}^{A}[\cdot]$ program is presented in Figure 4.7. The shell runs with ID $(\mathsf{eid}||c, \mathsf{idx})$ as a subroutine to the top level shell $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$, which implements the full oracle set, including the attack Rollback $\in \mathbb{A}$. Whenever the inner shell calls to a feature oracle, its execution is paused by $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$, which computes the oracle value and writes it on the subroutine output tape. Shell $(\mathsf{eid}||c, \mathsf{idx})$ is oblivious to this mechanism, and can simply call the oracles as if they were local subroutines. The identity of the shell includes counter c because the shell is one of the copies created by the shell $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ from Section 4.3.4 to enable rollbacks. All shell copies created for new RESUME iterations share the same storage interface for Store, Fetch. $(\mathsf{eid}||c, \mathsf{idx})$ instantiates a subroutine $(\mathsf{prog}, (\mathsf{eid}||c||\text{“wrapped”}, \mathsf{idx}))$ to execute the code of prog. For most of the execution of prog, it allows the internal subroutine to run. Since the execution of $(\mathsf{eid}||c, \mathsf{idx})$ is also running within an execution loop of $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$, whenever $(\mathsf{prog}, (\mathsf{eid}||c||\text{“wrapped”}, \mathsf{idx}))$ calls an oracle, $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$ will pause the execution of both subroutines to provide a return value. Likewise, if the adversary issues an Abort attack, $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$ will handle it directly.

Our protocol provides an inc-then-store counter (see Section 2.2.3.1) - despite the name, performing the store operation corresponds to a counter increase, since the local storage is reliable, and we do it before returning (storing) to the untrusted party.

To show that Conjecture 4.2 holds for the above protocol, we need to show that the protocol UC-emulates a G'_{att} functionality without Rollback. To prove this, we construct a simulator that turns any attempt at a Rollback into an Abort. Our simulator roughly follows the sketch outlined in the first case of the proof strategy for Conjecture 4.1, although we modify it appropriately for the adversarial case.

Assume the simulator has access to the same parameters as $\mathcal{W}$. The simulation

```
Shell $\mathrm{W}^A[\mathsf{prog}, \mathrm{nextmsg}_{\mathrm{meas}}]$

The identity of the shell is $(\mathsf{eid} \mid\mid c, \mathsf{idx})$
The parent shell extended identity is $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathrm{W}^{\mathrm{I}}[\mathsf{prog}]], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$

On message INIT from $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$:
  if $\mathrm{Fetch}() \neq \varepsilon$ then
      return ABORT
  install virtual ITI $(\mathsf{prog}, (\mathsf{eid}||c||\text{“wrapped”}, \mathsf{idx}))$
  let $m \leftarrow \mathrm{Meas}()$
  $\mathrm{Store}(m)$
On message input from $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$:
  while $\top$ do
      $\mathsf{out} \leftarrow \mathrm{nextmsg}_{\mathrm{meas}}(\text{tapes of virtual ITI})$
      if $\mathsf{out} = (\mathsf{pid}, \mathsf{eid}, (\textsc{MeasExec}, \mathsf{input}))$ then
          run $\textsc{MeasExec}(\mathsf{input})$
      else if $\mathsf{out} = (\perp, \perp, v) \wedge v \neq \text{“abort''}$ then
          return $v$
      else
          erase the virtual ITI work tape
          abort
On message $(\textsc{MeasExec}, \mathsf{input})$:
  let $m \leftarrow \mathrm{Fetch}()$
  let $m' \leftarrow \mathrm{Meas}()$
  if $m \neq m'$ then abort
  begin executing input on $(\mathsf{prog}, (\mathsf{eid}||c||\text{“wrapped”}, \mathsf{idx}))$
  for next instruction i on virtual ITI do
      if $\mathsf{i} = (\textbf{return } v)$ then
          $b \leftarrow \mathrm{Write}(\mathrm{Meas}())$
          assert $b = OK$
      else allow $(\mathsf{prog}, (\mathsf{eid}||c||\text{“wrapped”}, \mathsf{idx}))$ to execute $i$
```

Figure 4.7: The $\mathrm{W}^A[\cdot]$ enclave shell installed by protocol $\mathcal{W}$ for rollback iteration c of enclave eid installed by party pid for session idx

translates all requests to install a wrapped enclave from corrupted parties into requests to install the unwrapped enclave in G'_{att}; any attempt to install an unwrapped enclave will be simulated "in its head". Thereafter, whenever the party resumes one of the wrapped enclaves, the simulator fakes an access to the Fetch oracle, to reproduce the behaviour of the MEASEXEC subroutine in wrapper $\mathrm{W}^{A}[\cdot]$ to check that the enclave was not previously rolled back. If the check succeeds, the enclave begins executing the program ideally through running its non-wrapped version through the G'_{att} functionality. During its execution, the shell $(\mathsf{sh}_{\mathbb{O},\mathbb{A}'}[\mathsf{prog_w}], (\mathsf{eid}||\mathsf{pid}, \text{"att"}||\mathsf{idx}))$ might send messages on the backdoor tape related to some attacks in $\mathbb{A}'$ unrelated to rollback (therefore present in both real and ideal world). In that case, the simulator forwards it to the adversary and returns its response back to the shell without modification. After the execution of the enclave program has terminated, the simulator fakes a call to the Store oracle, with the length of the hash function used for mesuring enclave states (m) as its leakage.

If at any point during the simulation the adversary aborts a simulated oracle call, or if the simulator has recorded in dictionary P that the adversary has issued a rollback attack against that enclave, it will issue an abort message through the adversarial interface of G'_{att}, and halt its own execution. Otherwise, if all the checks succeed, it returns the output value and attestation signatures produced by G'_{att}. Additionally, the simulator produces an ITER message to signal that the resume execution has been successful, and the creation of a new copy for the ITI state (as if the enclave was running on G_{att}). Attestation verification requests are forwarded to G'_{att} if they are for the wrapped version of an enclave (where it will succeed only if the unwrapped version of the same enclave issued that message, before being transformed by F). Any request to verify a message where the attestation contains the unwrapped code (which is what is actually running on G'_{att}) is rejected.

Calls to install, resume, or verify the attestation of any unwrapped enclaves are not allowed by the protocol, but a corrupted party might try to get around this by directly writing to the tapes of real world G_{att} subroutine - this is allowed by the identity bound. In that case, the simulator lets the message through to its local simulated G_{att} subroutine, which can produce a convincing attestation signature for any message by using the original s algorithm. To denote this, we adopt the convention of forwarding adversarial messages for unwrapped enclaves to a "fake" copy of the hybrid functionality G^{F}_{att}. It is possible to think of G^{F}_{att} as simply shorthand for the book keeping operations inlined by the simulator's code, similar to the roles of the dictionary $\mathcal{G}$ in the Steel simulator of

Section 3.4, Alternatively, it is possible to see G^F_{att} as a bona-fide instance of G^{mod}_{att} run by the simulator as a local subroutine, and therefore granting no access to machines in other sessions. Adopting this view is only possible in our modular setting: while the Steel simulator, in the presence of G^{PST}_{att}, was required to keep a separate record of all messages signed by adversarial enclaves, this is the default for G^{mod}_{att}, and therefore we do not require keeping track of any additional operations. G^F_{att} is taken to be initialised with the same arguments as the real world G_{att} emulated by the protocol, such that any attempts to access an attack in $\mathbb{A}$ is reproduced by its (simulated) shells.

The pseudocode for the shell described above is as follows:

Simulator $\mathcal{S}$

$F(a,f)$ is the function that transforms an attestation measurement a so that it replaces the code of an enclave program p with code $\mathrm{W}^A[p,f]$. $\mathbb{M}$ is the standard uniform length for the output of Meas() oracle calls

State variables	Description
$\mathrm{P} \leftarrow []$	List of state pointers for rollback protected enclaves

```
On message INITIALISE from $G'_{\mathsf{att}}$:
  send INITIALISE to $\mathcal{A}$ through $G_{\mathsf{att}}$ and receive $pk, s$
  $\mathrm{nextmsg_{meas}} \leftarrow \mathsf{map^L}(\bar{\mathrm{E}}^R)$
  let $s'(x) \leftarrow s(F(x, \mathrm{nextmsg_{meas}})$
  send $(pk, s'))$ to $G'_{\mathsf{att}}$ on behalf of $\mathcal{A}$
  send INITIALISE to $G^F_{\mathsf{att}}$ through $\mathcal{Z}$ and receive INITIALISE
  send $\Sigma$ to $G^F_{\mathsf{att}}$ on behalf of $\mathcal{A}$
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from corrupted party P:
  if $\mathsf{prog} = \mathrm{W}^A[\mathsf{prog_w}, \mathrm{nextmsg_{meas}}]$ then
    send $(\text{INSTALL}, \mathsf{prog_w})$ to $G'_{\mathsf{att}}$ through $P$ and receive eid
    $\mathrm{P}[P, \mathsf{idx}, \mathsf{eid}, \mathsf{prog_w}] \leftarrow (\emptyset, \emptyset)$
  else
    send $(\text{INSTALL}, \mathsf{prog})$ to $G^F_{\mathsf{att}}$ through $P$ and receive eid
  return eid
On message $(\text{RESUME}, \mathsf{eid}, (i \parallel \mathsf{input}), \mathrm{Rollback})$ from corrupted party P:
  if $\mathrm{P}[P, \cdot, \mathsf{eid}, \cdot] = (c, c_{latest})$ then
    $\mathrm{P}[P, \cdot, \mathsf{eid}, \cdot] \leftarrow (i, c_{latest})$
    if $\mathsf{input} \neq \varepsilon$ then
```

```
        run $\mathsf{out}, \sigma \leftarrow \textsc{resume}(\mathsf{eid}, \mathsf{input}, \varepsilon)$,
    else
        send $(\textsc{Iter}, c, i)$ to $\mathcal{A}$ on behalf of $(\mathsf{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
  else send $(\textsc{resume}, \mathsf{eid}, (i \mid\mid \mathsf{input}), \mathrm{Rollback})$ to $G^F_{\mathsf{att}}$ on behalf of $P$
On message $(\textsc{resume}, \mathsf{eid}, \cdot, \mathrm{Abort})$ from corrupted party $P$:
  if $(\cdot,\cdot) \in \mathrm{P}[P, \cdot, \mathsf{eid}, \cdot]$ then send $(\textsc{resume}, \mathsf{eid}, \varepsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ on behalf of $P$
  else send $(\textsc{resume}, \mathsf{eid}, \varepsilon, \mathrm{Abort})$ to $G^F_{\mathsf{att}}$ on behalf of $P$
On message $(\textsc{resume}, \mathsf{eid}, \mathsf{input}, a)$ from corrupted party $P$:
  if $(c, c_{latest}) \in \mathrm{P}[P, \cdot, \mathsf{eid}, \mathsf{prog}_{\mathsf{w}}]$ then
    assert $a = \varepsilon \vee a \in \mathbb{A}'$
    let $shEID \leftarrow (\mathsf{sh}_{\mathbb{O},\mathbb{A}'}[\mathsf{prog}_{\mathsf{w}}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
    send $\textsc{Fetch}$ to $\mathcal{A}$ through $shEID$ and receive $b$
    if $b \neq \mathrm{Continue} \vee c \neq c_{latest}$ then send $(\textsc{resume}, \mathsf{eid}, \varepsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ and return
    send $(\textsc{resume}, \mathsf{eid}, \mathsf{input}, a)$ to $G'_{\mathsf{att}}$ on behalf of $P$ and
    while receive $(\mathrm{msg}, args)$ from $shEID$ do
        send $(\mathrm{msg}, args)$ to $\mathcal{A}$ through $shEID$ and receive $\textsc{response}$
        send $\textsc{response}$ to $shEID$ on behalf of $\mathcal{A}$
        if $\textsc{response} =$ Abort then return
    receive $\mathsf{out}, \sigma$ from $G'_{\mathsf{att}}$
    send $(\textsc{Store}, 1^{\mathbb{M}})$ to $\mathcal{A}$ through $shEID$ and receive $b'$
    if $b' \neq \mathrm{Continue}$ then send $(\textsc{resume}, \mathsf{eid}, \varepsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ and return
    generate nonce $c' \xleftarrow{\$} \{0,1\}^{\lambda}, \mathrm{P}[P, \cdot, \mathsf{eid}, \mathsf{prog}_{\mathsf{w}}] \leftarrow (c', c')$
    send $(\textsc{Iter}, c, c')$ to $\mathcal{A}$ on behalf of $shEID$
  else
    send $(\textsc{resume}, \mathsf{eid}, \mathsf{input}, \varepsilon)$ to $G^F_{\mathsf{att}}$ and receive $\mathsf{out}, \sigma$
  return $\mathsf{out}, \sigma$
On message $(\textsc{verify}, \sigma, m)$ from corrupted party $P$:
  if $m$ is a measurement for an enclave with program $\mathrm{W}^A[\mathsf{prog}_{\mathsf{w}}, \mathrm{nextmsg}_{\mathrm{meas}}]$ then
    send $(\textsc{verify}, \sigma, F(m, \mathrm{nextmsg}_{\mathrm{meas}}))$ to $G'_{\mathsf{att}}$ and receive $v$
  else if $m$ is a measurement for a program $\mathsf{prog}$ with enclave ID $\mathsf{eid}$ installed by some
  party $P'$ in session $\mathsf{idx}$, and $\mathrm{P}[P', \mathsf{idx}, \mathsf{eid}, \mathsf{prog} \neq \perp]$ then
    return $\perp$
  else
    send $(\textsc{verify}, \sigma, m))$ to $G^F_{\mathsf{att}}$ and receive $v$
  return $v$
```

For any protocol that adopts the standard identity bound, preventing the environment from sending messages on behalf of corrupted parties outside of the test session, the environment can not distinguish the real or ideal world, due to the simulator constructing a perfect transcript for the execution of $\mathcal{W}$ with the attestation signatures in the ideal world verifying for a real world $\mathrm{W}^{A}[\cdot]$ program.

Consider the case where the adversary does not conduct a rollback attack. For every RESUME operation from the corrupted party, the simulator activates the adversary with message FETCH, allowing it to interrupt the computation. If this happens, the simulator mounts the equivalent ABORT attack on G'_{att}. If FETCH is allowed, the measurement stored will be the same as from the previous execution, and therefore the simulator runs the program in G'_{att}. The behaviour of this execution is equivalent to the real world setup, since the shells of G_{att} and G'_{att} implement the same (non-rollback) oracles, and the simulator lets through any such adversarial access. Finally, the adversary receives a final STORE for a message of the same length as a MEAS value. Since the storage oracle does not leak the message contents but only their size, the adversary can not distinguish it from a state storage as executed during the MEASEXEC subroutine. If it chooses to abort, the real world wrapper would never terminate, so the simulator does the same for the ideal world enclave (by issuing its own ABORT), otherwise it returns the (ideally computed) value. The distribution of the return value for the enclave as executed in G_{att} and G'_{att} is equivalent (given they have the same feature oracles implementation), and the modified signature scheme attests to code $\mathrm{W}^{A}[\mathsf{prog}, f]$ in both worlds, thanks to the transformation F.

For the case of an adversary who, after some sequence of successful resumes, issues a rollback attack to an earlier state. The code of subroutine $\mathrm{W}^{A}[\cdot]$ does not allow executing any further RESUME, since the assertion that the measurement stored is equal to the current one will fail with non-negligible probability (as long as the measurement computed by oracle MEAS is collision-resistant, and the code of the enclave program iterates through a sufficiently diverse state distribution[3]). The simulator perfectly reproduces this behaviour, by issuing an ABORT to the ideal enclave, after having issued the preceding FETCH.

[3]If the enclave is running a program with a very limited set of states, such as a small finite state automaton, it is possible to artificially expand the state space by augmenting the program with a monotonically increasing counter for each resume. This will ensure that every measurement is distinct.

Chapter 5

Glass-Vault: A Generic Accountable Privacy-preserving Exposure Notification Analytics Platform

[A]ssuming security of hardware is neither cryptographically interesting nor really satisfying in a cryptography paper.

Reviewer #3

To conclude this work, we provide an example of a privacy-preserving protocol that is made practically efficient by using Trusted Execution Environments, and can be proved secure under Universal Composability by re-using existing secure protocols as its component.

GlassVault is an extension of regular privacy-preserving decentralised contact tracing that allows infected users to share sensitive (non-contact tracing) data for analysis.

As a *generic* platform, GlassVault is (data) type agnostic and supports *any secure computation* authorised by users. It offers accountability and privacy, by allowing users to consensually choose what data to share, and only allows analysts to execute computations authorised by sufficiently many users, while learning nothing beside the result about the input of users.[1]

In this work, we formally define GlassVault in the Universal Composability (UC) framework and prove its security. We consider a setting where both analysts and users may collude with each other to learn additional information about the data beyond what the analysts are authorised to compute. Furthermore, we allow the adversary to adaptively corrupt users, but assume a static set of corrupted analysts.

For privacy, it is sufficient that an upper bound for the user corruption threshold is publicly known to users when they encrypt their data. Here we also implicitly assume that data often looses its value and sensitivity over time. Thus, users might accept that the threshold could be reached far in the future as more users join the protocol and begin approving more functions. To have an efficient protocol that can offer all the above features, we construct GlassVault by carefully combining pre-existing exposure notification algorithms with an extension of the generalised functional encryption scheme proposed in Chapter 3. We extend their construction into a novel protocol called *DoubleSteel*, which allows a functional key to be generated in a distributed fashion while letting legitimate users freely join key authorisation and generation committees. We believe DoubleSteel to be of independent interest for other data analytic applications. While GlassVault is constructed by using an exposure notification protocol that reflects the implementation of popular decentralised contact tracing schemes deployed by many countries during the COVID-19 pandemic, its ability to allow analysts to compute a function on user-uploaded data reflects the ability of a health authority to analyse the graph of user contacts in a centralised contact tracing scheme. We claim

[1]Authorisation here does not necessarily need to be taken literally, the average user might just approve requests by certain analysts by default without a user-interface prompt, but the fact that users are in the loop nevertheless is crucial for accountability as it facilitates involvement of privacy advocacy organisations.

that GlassVault resolves the inherent conflict between these paradigms, by implementing the contact graph as a GlassVault function. Additionally, as an additional application of GlassVault, we show how it can be utilised to help the analyst identify infection clusters through heatmaps.

Chapter Organisation Section 5.1 presents an overview of related work and provides key concepts we rely on. Section5.2 presents a formal definition of our new functional encryption variant, called *DD-FESR*, along with DoubleSteel, a protocol for realising it using trusted hardware. Section5.3 presents the formal definition in the form of an ideal functionality, the protocol, and the security proof for GlassVault. Section **??** addresses how to bridge the divide between decentralised and decentralised contact tracing through GlassVault. Section5.5 gives the infection heatmaps example.

5.1 An Overview of Privacy-Preserving Contact Tracing systems

In this section, we present an overview of privacy-preserving solutions for contact tracing and exposure notification.

5.1.1 Centralised vs Decentralised Contact Tracing.

Within the first few months of the COVID-19 pandemic, a large number of theoretical (e.g., [30, 236, 279, 83]) and practical (e.g., [13, 151, 12, 3, 15]) automated contact tracing solutions were quickly developed by governments, industry, and academic communities. Most designs concentrated around two architectures, so-called *centralised* and *decentralised*, where the main difference between the two is that centralised systems keep a central record of the information of all users, regardless of infection status.

More technically, the major difference between the two architectures rests on key generation and exposure notification. In a centralised system, the keys are generated by a trusted health authority and distributed to contact tracing users, while also keeping a central user record. In decentralised systems, the keys are generated locally by each user. In both types, information is exchanged in a peer-to-peer fashion, commonly through Bluetooth Low Energy (BLE) messages broadcast by each user's phone. Once someone is notified of an infection, they upload some data to the health authority server. While in centralised systems the uploaded data usually corresponds to the BLE broadcast observed by infected user's devices, in a decentralised system it will generally be the messages they sent. Since the (centralised) authority keeps a record for each party in the system, it can identify their BLE broadcast of contacts and notify them about exposure. In this setting, the authority is able to construct users' contact graphs, which allows it to further analyse users' movements and interactions, at the cost of privacy.

On the other hand, the users of a decentralised system download the list of broadcasts from exposed users and compare it with their own local lists. This guarantees additional privacy compared to centralised systems (although several attacks are still possible, see [198]), while also preventing the health authority from running large scale analysis on infection data which would be possible in a centralised system. Despite this, the adoption of decentralised systems such as DP-3T [280] has been more widespread due to technical restrictions and political decisions forced by smartphone manufacturers [24]. There has been much debate on how any effort to the adoption of a more private and featureful contact tracing scheme could be limited by these gatekeepers [287, 30, 288].

5.1.2 Automated Data Analysis.

A few attempts have been made to develop automated systems which can analyse population behaviour to better understand the spread of the virus. The solution proposed in Bruni et al. [66] displays the development of virus hotspots, as a heatmap. In this system, there are two main players; namely, the health authority and a mobile phone provider, each of which has a set of data that they have independently collected from their users. Their goal is to find (only) the heatmap in a privacy-preserving manner, i.e., without revealing their input in plaintext to their counterpart. To achieve its goal, the system uses (computationally expensive) homomorphic encryption, differential privacy, and a matrix representation of inputs. In this system, (i) the two parties run the computation on users' data, without having their fine-grained consent and (ii) each party's input has to be encoded in a specific way, i.e., it must be encrypted and represented as a matrix.

The protocols in [191, 55, 149] allow users to provide their encoded data to a server for a specific analysis. Specifically, Lueks et al. [191] design a privacy-preserving

Presence-Tracing system which notifies people who were in the same venue as an infected individual. The proposed solution mainly uses identity-based encryption, a hash function, and authenticated encryption to achieve its goal and encode users' input data. Biasse et al. [55] design a privacy-preserving scheme to anonymously collect information about a social graph of users. In this solution, a central server can construct an anonymous graph of interactions between users which would let the server understand the progression of the virus among users. This solution is based on zero-knowledge proofs, digital signatures, and RSA-based accumulators. Günther et al. [149] propose a privacy-preserving scheme between multiple non-colluding servers to help epidemiologists simulate and predict future developments of the virus. This scheme relies on heavy machinery such as oblivious shuffling, anonymous credentials, and generic multi-party computation.

In all of the above solutions, the parties need to encode their inputs in a certain way to support the specific computation that is executed on their inputs, and thus do not support generality. Additionally, not all systems allow the users to opt-out of the computation, and are therefore not fully accountable to them.

5.1.3 Formalising Exposure Notification in the UC framework.

Canetti et al. [84] introduce a comprehensive approach to formalise the Exposure Notification primitive via the UC framework, showing how a protocol similar to DP-3T realises their ideal functionality. Their UC formulation is designed to capture a wide range of Exposure Notification settings. The modelling relies on a variety of functionalities that abstract phenomena such as physical reality events and Bluetooth communications. While the above work is unique in formalising Exposure notification, a UC formalisation of the related problem of proximity testing has been given in [275], based on the reduction to Private Equality Testing in [214].

We now provide a full specification of the Exposure Notification functionality, including some additionally prerequisites.

5.1.3.1 The repository functionality $\mathcal{REP}$

We relax the functionality $\mathcal{REP}$ from Section 2.3.4 by allowing any party to read/write, as long as the read/write request refers to some specified session. Namely, the functionality keeps a table M of the all the messages submitted by writing requests. On message $(\textsc{write}, x)$ from W, it generates a unique handle h, and records x in $M[\mathsf{h}]$. On message $(\text{READ}, \mathsf{h})$ from a party $P \in \mathbf{R}$, it returns $M[\mathsf{h}]$ to P.

5.1.3.2 The time functionality $\mathbb{T}$

The time functionality $\mathbb{T}$ of [84] can be used as a clock within a UC protocol. It initializes a counter t as 0. On message INCREMENT from the environment, it increments t by 1. On message TIME from a party P , it sends t to P.

5.1.3.3 The physical reality functionality $\mathbb{R}$

Functionality $\mathbb{R}$ introduced in [84], represents the "physical reality" of each participant to a protocol, meaning the historical record of all physical facts (e.g., location, motion, visible surroundings) involving the participants.
$\mathbb{R}$ is parameterized by a validation predicate V for checking that the records provided by the environment are sensible, and a set $\mathbf{F}$ of ideal functionalities that have full access to the records obtained by $\mathbb{R}$. The functionality only considers records that have a specific format and include the party identity, time, and the types of measurement (e.g., location, altitude, temperature, distance of the party from each other party, health status) that evaluate the physical reality for the said party. It initializes a list R of all submitted records that are in correct format and operates as follows:

- On message (P, v) from the environment, where P is a party's identity and v is a record in correct format, it appends (P, v) to R. Then, it sends TIME to $\mathbb{T}$ (the time functionality presented in 5.1.3.2) and obtains t. It checks that t matches the time entry in v and that $V(R)$ holds. If any check fails, then it halts.
- On message $(\text{MyCurrentMeas}, P, L, e)$ that comes from either party P or a functionality in $\mathbf{F}$ (otherwise, it returns an error), where L is a list of fields that refer to the correct record format and e is an error function:
  1. It finds the latest entry v in the sub-list of entries in R whose first element is P.
  2. It sets v_L as the record v restricted to the fields in L.
  3. It computes $e(v_L)$, i.e., the result of applying the error function e to v_L.
  4. It returns $e(v_L)$.
- On message $(\text{AllMeas}, e)$ from a functionality in $\mathbf{F}$, it applies e to each record in R and obtains $\tilde{R}$. It returns $\tilde{R}$.

5.1.3.4 The trusted bulletin board functionality $\mathcal{F}_{\mathsf{TBB}}$

The functionality $\mathcal{F}_{\mathsf{TBB}}$, as presented in [84], maintains a state that is updated whenever new data are uploaded (for infectious parties). It initializes a list $\mathcal{C}$ of records. On message $(\textsc{Add}, c)$ from a party P, it checks with $\mathbb{R}$ whether P is infectious (formally, $\mathcal{F}_{\mathsf{TBB}}$ sends a message $(\textsc{MyCurrentMes}, P,$ "health_status"$, \mathsf{id})$ to $\mathbb{R}$, where id is the identity function). If this holds, then it appends c to $\mathcal{C}$. On message $\textsc{Retrieve}$ from a party P, it returns $\mathcal{C}$ to P.

5.1.3.5 The exposure notification functionality $\mathcal{F}_{\mathrm{EN}}$

The Exposure Notification functionality, also introduced in [84], builds on the previous two functionalities to provide a mechanism for warning people who have been exposed to infectious carriers of the virus. The description of the functionality is recapped in section 5.3.1; we show the formal description for the purposes of comparing this functionality with $\mathcal{F}_{\mathrm{EN}^+}$.

Confirmation of test results when sharing exposure and re-registration into the system for no longer infectious users is not captured by the functionality.

Functionality $\mathcal{F}_{\mathrm{EN}}[\rho, E, \Phi, \mathcal{L}, \mathcal{P}]$

State variables	Description
SE	List of users who have shared their exposure status
$\overline{\mathbf{U}}$	List of active users
$\widetilde{\mathbf{U}}$	List of corrupted users
$\widetilde{\mathrm{R}}_\varepsilon$	Noisy record of physical reality

```
On message (SETUP, $\varepsilon^*$) from $\mathcal{A}$:
  assert $\varepsilon^* \in E$;  $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \emptyset$
On message ACTIVATEMOBILEUSER from $U \in \mathcal{P}$:
  $\overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \parallel U$
  send (ACTIVATEMOBILEUSER, $U$) to $\mathcal{A}$
On message SHAREEXPOSURE from $U \in \mathcal{P}$:
  send (ALLMEAS, $\varepsilon^*$) to $\mathbb{R}$ and receive $\widetilde{\mathrm{R}}^*$
  $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \widetilde{\mathrm{R}}_\varepsilon \parallel \widetilde{\mathrm{R}}^*$
  if $\widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{INFECTED}] = \perp$ then
    return error
    else
        send TIME to $\mathbb{T}$ and receive $t$
        $\mathbf{SE} \leftarrow \mathbf{SE} \,\|\, (U,t)$
        $\overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \setminus \{U\}$
        send $(\textsc{ShareExposure}, U)$ to $\mathcal{A}$
On message ExposureCheck from $U \in \mathcal{P}$:
    if $U \in \overline{\mathbf{U}}$ then
        send $(\textsc{AllMeas}, \varepsilon^*)$ to $\mathbb{R}$ and receive $\widetilde{\mathrm{R}}^*$
        $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \widetilde{\mathrm{R}}_\varepsilon \,\|\, \widetilde{\mathrm{R}}^*$
        $\mu \leftarrow \widetilde{\mathrm{R}}_\varepsilon[U] \,\|\, \widetilde{\mathrm{R}}_\varepsilon[\mathbf{SE}]$
        return $\rho(U,\mu)$
    else return error
On message RemoveMobileUser from $U \in \mathcal{P}$:
    $\overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \setminus \{U\}$
On message $(\textsc{Corrupt}, U)$ from $\mathcal{A}$:
    $\widetilde{\mathbf{U}} \leftarrow \widetilde{\mathbf{U}} \,\|\, U$
On message $(\textsc{MyCurrentMeas}, U, A, e)$ from $\mathcal{A}$:
    if $U \in \widetilde{\mathbf{U}}$ then
        send $(\textsc{MyCurrentMeas}, U, A, e)$ to $\mathbb{R}$ and receive $u_A^e$
        send $(\textsc{MyCurrentMeas}, u_A^e)$ to $\mathcal{A}$
On message $(\textsc{FakeReality}, \phi)$ from $\mathcal{A}$:
    if $\phi \in \Phi$ then
        $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \phi(\widetilde{\mathrm{R}}_\varepsilon)$
On message Leak from $\mathcal{A}$:
    send $(\textsc{Leak}, \mathcal{L}(\{\widetilde{\mathrm{R}}_\varepsilon, \overline{\mathbf{U}}, \mathbf{SE}\}))$ to $\mathcal{A}$
On message $(\textsc{IsCorrupt}, U)$ from $\mathcal{Z}$:
    return $U \overset{?}{\in} \widetilde{\mathbf{U}}$
```

5.2 Dynamic and Decentralised FESR (DD-FESR) and Steel (DoubleSteel)

In this section, we present the ideal functionality DD-FESR and the protocol that realises it, DoubleSteel. As we stated earlier, they are built upon the original functionality FESR and protocol Steel, respectively. In the formal descriptions, we will highlight the main changes that we have applied to the original scheme from Chapter 3 in yellow. As the construction is of independent interest from the contact-tracing problem, we refer to the various parties in the protocol with their generic roles of encryptors and decryptors. In the next section, we will use DD-FESR as a subroutine, and define which Exposure Notification parties fulfill each role. For conciseness, we omit any reference to UC-specific machinery, such as session IDs, unless it is required to understand the specifics of our protocol.

5.2.1 The Ideal Functionality DD-FESR

In this subsection, we extend the functional encryption scheme FESR into a new functionality DD-FESR to capture two additional properties from the functional encryption literature:

1. *Decentralisation* (introduced in [96]), which allows a set of encryptors to be in control of functional key generation, rather than a single trusted authority.
2. *Dynamic* membership (introduced in [97]) allows any party from a known legitimate party set $\mathcal{P}$ to freely join sets $\mathbf{A}$ (encryptors) and/or $\mathbf{B}$ (decryptors) during the execution of the protocol.

Functionality Overview. As in Chapter 3, our definition of DD-FESR builds on the work of [202] in casting the Functional Encryption scheme as an access controlled repository, where encryptors "upload" data and decryptors "download" it. In our variants, encryptors also vote on a "download policy" by deciding which decryptor is able to access what function over the uploaded data.

The functionality DD-FESR registers parties from $\mathcal{P}$ as encryptors and/or decryptors, when they provide DD-FESR with an according SETUP message.

The KEYGEN subroutine of the FESR scheme is replaced by a new subroutine KEYSHAREGEN. Namely, any party $\mathsf{A} \in \mathbf{A}$ informs DD-FESR that she allows a *key share* for a decryptor B w.r.t. some function F. In turn, the functionality provides B with a $(\text{KEYSHAREGEN}, \mathrm{F}, \mathsf{A}, \mathsf{B})$ message.

Upon an encryption request of a message x under a decryption threshold k by any registered party, the functionality selects a unique index h and records an associated (x, k). Then, it responds to the party with $(\text{ENCRYPTED}, \mathsf{h})$.

When a decryptor B requests a decryption w.r.t. an index h and a function F, then B must have collected at least k shares for F before decryption is allowed. In particular, DD-FESR recovers the pair (x, k) associated with h and checks if it has issued at least k $(\text{KEYSHAREGEN}, \mathtt{F}, \cdot, \mathsf{B})$ messages to B. If so, it chooses randomness r, and runs the stateful computation of F on input x and randomness r, which results in the output y. Then, it sends $(\text{DECRYPTED}, \mathsf{y})$ to B.

The functionality is presented in detail below.

Functionality DD-FESR$[\mathtt{F}, \mathcal{P}]$

The functionality is parameterised by the randomised function class $\mathtt{F} = \{\mathrm{F} \mid \mathrm{F} : \mathcal{X} \times \mathcal{S} \times \mathcal{R} \to \mathcal{Y} \times \mathcal{S}\}$, over state space $\mathcal{S}$ and randomness space $\mathcal{R}$, and by a set of (dummy) parties $\mathcal{P}$.

State variables	Description
$\mathbf{A} \leftarrow []$	List of registered encryptors
$\mathbf{B} \leftarrow []$	List of registered decryptors
$\hat{\mathbf{A}} \leftarrow []$	List of corrupted encryptors
$\hat{\mathbf{B}} \leftarrow []$	List of corrupted decryptors
F_0	Leakage function returning message length
$\mathtt{F}^+$	$\mathtt{F} \cup \mathrm{F}_0$
$\mathsf{setup}[\cdot] \leftarrow \mathsf{false}$	Table recording which parties were initialised.
$\mathcal{M}[\cdot] \leftarrow \perp$	Table storing the plaintext for each message handler
$\mathcal{P}[\cdot] \leftarrow \emptyset$	Table of authorised functions' states for all decryption parties
$\mathcal{KS}[\cdot] \leftarrow []$	Table of key share generator for each (decryptor, function) pair

On message $(\text{SETUP}, \mathsf{role})$ *from* $P \in \mathcal{P}$:

```
    assert $\mathsf{setup}[P] = \mathsf{false}$
    send $(\text{SETUP}, \mathsf{role}, P)$ to $\mathcal{A}$ and receive OK
    if $P \in \mathcal{P} \setminus \mathbf{A} \wedge \mathsf{role} = \mathsf{encryptor}$ then $\mathbf{A} \leftarrow \mathbf{A} \parallel P$
    else if $P \in \mathcal{P} \setminus \mathbf{B} \wedge \mathsf{role} = \mathsf{decryptor}$ then $\mathbf{B} \leftarrow \mathbf{B} \parallel P$
    else return
    $\mathsf{setup}[P] \leftarrow \mathsf{true}$
```

On message $(\text{KEYSHAREGEN}, \mathrm{F}, \mathsf{B})$ *from* $\mathsf{A} \in \mathbf{A}$:

```
    if $\mathsf{A} \in \hat{\mathbf{A}}$ then                ▷ The adversary can only block key generation for corrupted parties
        send $(\text{KEYSHAREQUERY}, \mathrm{F}, \mathsf{A}, \mathsf{B})$ to $\mathcal{A}$ and receive OK
    if $(\mathrm{F} \in \mathtt{F}^+ \wedge \mathsf{setup}[\mathsf{A}] \wedge \mathsf{setup}[\mathsf{B}])$ then
        $\mathcal{KS}[\mathsf{B}, \mathrm{F}] \leftarrow \mathcal{KS}[\mathsf{B}, \mathrm{F}] \parallel \mathsf{A}$        ▷ We store the identity of all parties in $\mathbf{A}$ who authorised F for B
        send $(\text{KEYSHAREGEN}, \mathrm{F}, \mathsf{A}, \mathsf{B})$ to $\mathsf{B}$
        if $\mathsf{B} \in \hat{\mathbf{B}} \vee \mathsf{A} \in \hat{\mathbf{A}}$ then send $(\text{KEYSHAREGEN}, \mathsf{F}, \mathsf{A}, \mathsf{B})$ to $\mathcal{A}$
        else send $(\text{KEYSHAREGEN}, \perp, \mathsf{A}, \mathsf{B})$ to $\mathcal{A}$
On message $(\text{ENCRYPT}, \mathsf{x}, k)$ from party $P \in \mathbf{A} \cup \mathbf{B}$:
    if $(\mathsf{setup}[P] \wedge \mathsf{x} \in \mathcal{X} \wedge k \text{ is an integer})$ then
        generate nonce $\mathsf{h} \xleftarrow{\$} \{0,1\}^{\lambda}$
        $\mathcal{M}[\mathsf{h}] \leftarrow (\mathsf{x}, k)$
        send $(\text{ENCRYPTED}, \mathsf{h})$ to $P$
    else
        send $(\text{ENCRYPTED}, \perp)$ to $P$
On message $(\text{DECRYPT}, \mathsf{F}, \mathsf{h})$ from party $\mathsf{B} \in \mathbf{B}$:
    $(\mathsf{x}, k) \leftarrow \mathcal{M}[\mathsf{h}]; \mathsf{y} \leftarrow \perp;$
    if $\mathsf{F} = \mathsf{F}_0$ then
        $\mathsf{y} \leftarrow |\mathsf{x}|$
    else if $((|\mathcal{KS}[\mathsf{B},\mathsf{F}]| \geq k \wedge \forall \mathsf{A} \in \mathcal{KS}[\mathsf{B},\mathsf{F}] : \mathsf{setup}[\mathsf{A}] \wedge \forall \mathsf{A} \text{ are distinct }) \vee (\mathsf{B} \in \hat{\mathbf{B}} \wedge |\hat{\mathbf{A}}| \geq k))$
    then ▷ There are at least $k$ functional key shares, all generated by correctly setup parties, OR B and at least $k$ parties in A are corrupted
        $\mathsf{s} \leftarrow \mathcal{P}[\mathsf{B},\mathsf{F}]$
        $\mathsf{r} \xleftarrow{\$} \mathcal{R}$
        $(\mathsf{y}, \mathsf{s}') \leftarrow \mathsf{F}(\mathsf{x}, \mathsf{s}; \mathsf{r})$
        $\mathcal{P}[\mathsf{B},\mathsf{F}] \leftarrow \mathsf{s}'$
    return $(\text{DECRYPTED}, \mathsf{y})$
On message $(\text{CORRUPT}, P)$ from $\mathcal{A}$:
    if $P \in \mathbf{A} \cap \mathbf{B}$ then
        $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \parallel P; \hat{\mathbf{B}} \leftarrow \hat{\mathbf{B}} \parallel P$    ▷ The functionality needs to keep track of corrupted parties in A to ensure correctness
        return $\{(\mathsf{B},\mathsf{F}) | P \in \mathcal{KS}[\mathsf{B},\mathsf{F}]\}$ and $\mathcal{KS}[P, \cdot]$
    if $P \in \mathbf{A} \setminus \mathbf{B}$ then
        $\hat{\mathbf{A}} \leftarrow \hat{\mathbf{A}} \parallel P$
        return $\{(\mathsf{B},\mathsf{F}) | P \in \mathcal{KS}[\mathsf{B},\mathsf{F}]\}$
    if $P \in \mathbf{B} \setminus \mathbf{A}$ then
        $\hat{\mathbf{B}} \leftarrow \hat{\mathbf{B}} \parallel P$
        return $\mathcal{KS}[P, \cdot]$
```

5.2.2 The Protocol DoubleSteel

Now, we propose DoubleSteel, a new protocol that extends Steel to realise DD-FESR. We briefly provide an overview of Steel and our new extension, before outlining the formal protocol.

Overview of Steel. Steel uses a public key encryption scheme, where the master public key is distributed to parties in set $\mathcal{P}$, and the master secret key is securely stored in an enclave running program $\mathsf{prog}_{\mathsf{KME}}$ on trusted party C. The secret key is then provisioned to enclaves running on parties as decryptors, only if they can prove through remote attestation that they are running a copy of $\mathsf{prog}_{\mathsf{DE}}$. The functional key corresponds to signatures over the representation of a function F and is generated by C's $\mathsf{prog}_{\mathsf{KME}}$ enclave. If party B possesses any such key, its copy of $\mathsf{prog}_{\mathsf{DE}}$ will distribute the master secret key to a $\mathsf{prog}_{\mathsf{FE}[\mathrm{F}]}$ enclave, which can then decrypt any A's encrypted inputs x and will ensure that only value y of $(\mathrm{y},\mathrm{s}') \leftarrow \mathrm{F}(\mathrm{x},\mathrm{s};\mathrm{r})$ is returned to B, with the function states s and s' protected by the enclave.

Overview of DoubleSteel. The protocol DoubleSteel has a few crucial differences with respect to the original version as described. Party C, who is now untrusted, still runs the public key encryption parameter generation within an enclave and distributes it to a party A or B when they first join the protocol. Each party A also generates a digital signature key pair locally and includes a key policy k along with their ciphertext. Note that for simplicity our current version uses an integer k to associate with each message, but it would be possible to use a public key policy as a threshold version of Multi-Client Functional Encryption. Party A who wants to authorise party B to compute a certain function will run $\mathsf{KeyShareGen}(\mathrm{F},\mathsf{B})$ to generate a key share, which requires signing the representation of F with their local key, and send the signature to B. For B's $\mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}$ enclave to authorise the functional decryption of F, it first verifies that all key shares provided by B for F are valid and each was provided by a unique encryptor party; if all checks are passed, then it will distribute the master secret key to $\mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]}$, along with the length of recorded key shares k_{F}. $\mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]}$ will only proceed with decryption if the number of key shares meets the encryptor's key policy. Provisioning of the secret key between C and B's $\mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}$ enclave remains as in FESR.

The protocol DoubleSteel makes use of the global attestation functionality G_{att}, the certification functionality $\mathcal{F}_{\mathsf{CERT}}$, the common reference string functionality $\mathcal{CRS}$, the secure channel functionality $\mathcal{SC}_{S\to R}$, and the repository functionality $\mathcal{REP}$ that are presented in Sections 2.2.4.1,2.3.7, 2.3.5, 2.3.6, and 5.1.3.1 respectively. The code of the enclave programs $\mathsf{prog}_{\mathsf{KME}^{\mathsf{VK}}}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}, \mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\cdot]}$ in DoubleSteel is hardcoded with the value of the verification key VK returned by $\mathcal{F}_{\mathsf{CERT}}$, and can be generated during the protocol runtime.

Protocol $\mathsf{DoubleSteel}[\mathtt{F}, \mathsf{PKE}, \Sigma, \mathsf{N}, \lambda, \mathcal{P}, \mathsf{C}]$

The protocol is parameterised by the class of functions $\mathtt{F}$ as defined in DD-FESR, the public-key encryption scheme PKE denoted as the triple of algorithms $\mathsf{PKE} := (\mathsf{PGen}, \mathsf{Enc}, \mathsf{Dec})$, the digital signature scheme Σ denoted as the triple of algorithms $\Sigma := (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$, the non-interactive zero-knowledge protocol N that consists of prover $\mathcal{P}$ and verifier $\mathcal{V}$, and the security parameter λ. $\mathcal{P}$ is a set of legitimate parties of type A, B. C is the identity of the Key Generation party.

State variables	Description
$\mathcal{KS}[\cdot] \leftarrow \emptyset$	Table of function key shares for B
$\mathcal{K}[\cdot] \leftarrow \emptyset$	Table of functional enclave details for B

Key Generation Authority C:

```
On message (SETUP, role, P) from SC_{P→C}:
  if $\mathsf{mpk} = \bot$ then
    send GET to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
    send GETK to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive VK
    $\mathsf{eid}_{\mathsf{KME}} \leftarrow \mathcal{G}_{\mathsf{att}}.\mathsf{install}(\mathsf{C.sid}, \mathsf{prog}_{\mathsf{KME}^{\mathsf{VK}}})$
    $(\mathsf{mpk}, \sigma_{\mathsf{KME}}) \leftarrow \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathsf{init}, \mathsf{crs}, \mathsf{C.sid}))$
  if $P \in \mathcal{P} \wedge \mathsf{role} = \mathsf{encryptor}$ then
    send $(\textsc{Setup}, \mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}})$ to $\mathcal{SC}_{\mathsf{C}\to P}$
  else if $P \in \mathcal{P} \wedge \mathsf{role} = \mathsf{decryptor}$ then
    send $(\textsc{Setup}, \mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}})$ to $\mathcal{SC}_{\mathsf{C}\to P}$ and receive $(\textsc{Provision}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$
    $(\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}}) \leftarrow \mathcal{G}_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathsf{provision}, (\sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}})))$
    send $(\textsc{Provision}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}})$ to $\mathcal{SC}_{\mathsf{C}\to P}$
```

Party P as encryptor:

```
On message (SETUP, encryptor) from $\mathcal{Z}$:
  assert $\mathsf{mpk} = \bot$
  send $(\textsc{Setup}, \mathsf{encryptor})$ to $\mathcal{SC}_{P\to\mathsf{C}}$ and receive $\mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}}$
  send GETPK to $\mathcal{G}_{\mathsf{att}}$ and receive $\mathsf{vk}_{\mathsf{att}}$
  send GETK to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive VK
```

```
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, (\mathsf{sid}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}^{\mathsf{VK}}}, \mathsf{mpk}), \sigma_{\mathsf{KME}})$
    send GET to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
    $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
    send $(\textsc{Sign}, \mathsf{spk})$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive cert
    store $\mathsf{mpk}, \mathsf{crs}, \mathsf{spk}, \mathsf{ssk}, \mathsf{cert}$
  On message $(\textsc{KeyShareGen}, \mathsf{F}, \mathsf{B})$ from $\mathcal{Z}$:
    $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, \mathsf{F}, \mathsf{B})$
    send $(\textsc{KeyShareGen}, \mathsf{F}, \sigma, \mathsf{spk}, \mathsf{cert})$ to $\mathcal{SC}_{P \to B}$
  On message $(\textsc{Encrypt}, \mathrm{m}, k)$ from $\mathcal{Z}$:
    assert $\mathsf{mpk} \neq \perp \wedge \mathrm{m} \in \mathcal{X} \wedge k$ is an integer        ▷ Any party having mpk can encrypt
    $\mathsf{ct} \xleftarrow{r} \mathsf{PKE.Enc}(\mathsf{mpk}, (\mathrm{m}, k))$
    $\pi \leftarrow \mathcal{P}((\mathsf{mpk}, \mathsf{ct}), ((\mathrm{m}, k), \mathrm{r}), \mathsf{crs}); \mathsf{ct}_{\mathsf{msg}} \leftarrow (\mathsf{ct}, \pi)$
    send $(\textsc{write}, \mathsf{ct}_{\mathsf{msg}})$ to $\mathcal{REP}$ and receive h
    return $(\textsc{encrypted}, \mathsf{h})$
Party $P$ as decryptor:
  On message $(\textsc{Setup}, \mathsf{decryptor})$ from $\mathcal{Z}$:
    assert $\mathsf{mpk} = \perp$
    send $(\textsc{Setup}, \mathsf{decryptor})$ to $\mathcal{SC}_{P \to \mathsf{C}}$ and receive $\mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}}$
    $\mathcal{KS} = \{\}, \mathcal{K} = \{\}$
    send GETPK to $G_{\mathsf{att}}$ and receive $\mathsf{vk}_{\mathsf{att}}$
    send GETK to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive VK
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, (\mathsf{idx}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}^{\mathsf{VK}}}, \mathsf{mpk}), \sigma_{\mathsf{KME}})$
    store $\mathsf{mpk}; \mathsf{eid}_{\mathsf{DE}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}})$
    send GET to $\mathcal{CRS}$ and receive $(\textsc{Crs}, \mathsf{crs})$
    $((\mathsf{pk}_{KD}, \cdot, \cdot), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\text{init-setup}, \mathsf{eid}_{\mathsf{KME}}, \sigma_{\mathsf{KME}}, \mathsf{crs}, \mathsf{B.sid}))$
    send $(\textsc{provision}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$ to $\mathcal{SC}_{P \to \mathsf{C}}$ and receive
    $(\textsc{provision}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}})$
    $G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\text{complete-setup}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}))$
  On message $(\textsc{KeyShareGen}, \mathsf{F}, \sigma, \mathsf{spk}, \mathsf{cert})$ from $\mathcal{SC}_{\mathsf{A} \to P}$:
    $\mathcal{KS}[\mathrm{F}] \leftarrow \mathcal{KS}[\mathrm{F}] \,||\, (\sigma, \mathsf{spk}, \mathsf{cert})$
  On message $(\textsc{Decrypt}, \mathsf{F}, \mathsf{h})$ from $\mathcal{Z}$:
    if $\mathcal{K}[\mathrm{F}] = \perp$ then
      $\mathsf{eid}_{\mathrm{F}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]})$
      $(\mathsf{pk}_{\mathsf{DF}}, \sigma_{\mathrm{F}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{init}, \mathsf{mpk}, \mathsf{B.sid}))$
      $\mathcal{K}[\mathrm{F}] \leftarrow (\mathsf{eid}_{\mathrm{F}}, \mathsf{pk}_{\mathsf{DF}}, \sigma_{\mathrm{F}})$
    send $(\textsc{read}, \mathsf{h})$ to $\mathcal{REP}$ and receive $\mathsf{ct_{msg}}$
    $(\mathsf{eid_F}, \mathsf{pk_{DF}}, \sigma_F) \leftarrow \mathcal{K}[F]$
    $((\mathsf{ct_{key}}, k_F, \mathsf{crs}), \sigma_{\mathsf{DE}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{DE}}, (\mathrm{provision}, \mathcal{KS}[F], \mathsf{eid_F}, \mathsf{pk_{DF}}, \sigma_F, F, \mathsf{B.pid}))$
    $((\mathrm{computed}, y), \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_F}, (\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid_{DE}}, \mathsf{ct_{key}},$
    $\mathsf{ct_{msg}}, k_F, \mathsf{crs}, \perp))$
    return $(\textsc{decrypted}, y)$
$\underline{\mathsf{prog_{KME^{VK}}}}$
on input init
    $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{PKE.PGen}(1^\lambda)$
    return $\mathsf{pk}$
on input $(\mathrm{provision}, (\sigma_{\mathsf{DE}}, \mathsf{eid_{DE}}, \mathsf{pk}_{KD}, \mathsf{eid_{KME}}))$:
    $\mathsf{vk_{att}} \leftarrow G_{\mathsf{att}}.\mathsf{vk_{att}}$; fetch $\mathsf{crs}, \mathsf{idx}, \mathsf{sk}$
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk_{att}}, (\mathsf{idx}, \mathsf{eid_{DE}}, \mathsf{prog_{DE^{VK}}}, (\mathsf{pk}_{KD}, \mathsf{eid_{KME}}, \mathsf{crs}), \sigma_{\mathsf{DE}})$
    $\mathsf{ct_{key}} \leftarrow \mathsf{PKE.Enc}(\mathsf{pk}_{KD}, \mathsf{sk})$
    return $\mathsf{ct_{key}}$

$\underline{\mathsf{prog_{DE^{VK}}}}$
on input $(\mathrm{init\text{-}setup}, \mathsf{eid_{KME}}, \mathsf{crs}, \mathsf{idx})$:
    assert $\mathsf{pk}_{KD} \neq \perp$
    $(\mathsf{pk}_{KD}, \mathsf{sk}_{KD}) \leftarrow \mathsf{PKE.Gen}(1^\lambda)$
    store $\mathsf{sk}_{KD}, \mathsf{eid_{KME}}, \mathsf{crs}, \mathsf{idx}$
    return $\mathsf{pk}_{KD}, \mathsf{eid_{KME}}, \mathsf{crs}$
on input $(\mathrm{complete\text{-}setup}, \mathsf{ct_{key}}, \sigma_{\mathsf{KME}})$:
    $\mathsf{vk_{att}} \leftarrow G_{\mathsf{att}}.\mathsf{vk}$
    fetch $\mathsf{eid_{KME}}, \mathsf{sk}_{KD}, \mathsf{idx}$
    $m \leftarrow (\mathsf{idx}, \mathsf{eid_{KME}}, \mathsf{prog_{KME^{VK}}}, \mathsf{ct_{key}})$
    assert $\Sigma.\mathsf{Vrfy}(\mathsf{vk_{att}}, m, \sigma_{\mathsf{KME}})$
    $\mathsf{sk} \leftarrow \mathsf{PKE.Dec}(\mathsf{sk}_{KD}, \mathsf{ct_{key}})$
    store $\mathsf{sk}, \mathsf{vk_{att}}$
on input $(\mathrm{provision}, \mathcal{KS}_F, \mathsf{eid_F}, \mathsf{pk_{DF}}, \sigma_F, F, \mathrm{pid})$:
    fetch $\mathsf{eid_{KME}}, \mathsf{vk_{att}}, \mathsf{sk}, \mathsf{idx}, \mathsf{crs}$
    $m \leftarrow (\mathsf{idx}, \mathsf{eid}, \mathsf{prog_{FE^{VK}[F]}}, \mathsf{pk_{DF}})$
    assert $\forall(\sigma_{\mathsf{spk}}, \mathsf{spk}, \mathsf{cert}) \in \mathcal{KS}_F : (\Sigma.\mathsf{Vrfy}(\mathsf{VK}, \mathsf{spk}, \mathsf{cert}) \wedge$
    $\wedge \Sigma.\mathsf{Vrfy}(\mathsf{spk}, (F, \mathrm{pid}), \sigma_{\mathsf{spk}}) \wedge \forall \mathsf{spk}$ are distinct$)$ $\wedge \Sigma.\mathsf{Vrfy}(\mathsf{vk_{att}}, m, \sigma_F)$
    return $\mathsf{PKE.Enc}(\mathsf{pk_{DF}}, \mathsf{sk}), |\mathcal{KS}_F|, \mathsf{crs}$
```

```
$\underline{\mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]}}$
on input $(\mathrm{init}, \mathsf{mpk}, \mathsf{idx})$:
    **assert** $\mathsf{pk}_{\mathsf{DF}} = \perp$
    $(\mathsf{pk}_{\mathsf{DF}}, \mathsf{sk}_{\mathsf{DF}}) = \mathsf{PKE.Gen}(1^{\lambda})$
    $\mathsf{mem} \leftarrow \emptyset$; **store** $\mathsf{sk}_{\mathsf{DF}}, \mathsf{mem}, \mathsf{mpk}, \mathsf{idx}$
    **return** $\mathsf{pk}_{\mathsf{DF}}$
on input $(\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, k_{\mathrm{F}}, \mathsf{crs}, \mathrm{y}')$:
    **if** $\mathrm{y}' \neq \perp$
        **return** $(\mathrm{computed}, \mathrm{y}')$
    $\mathsf{vk}_{\mathsf{att}} \leftarrow \mathcal{G}_{\mathsf{att}}.\mathsf{vk}; (\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
    **fetch** $\mathsf{sk}_{\mathsf{DF}}, \mathsf{mem}, \mathsf{mpk}, \mathsf{idx}$
    $\mathrm{m} \leftarrow (\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}, (\mathsf{ct}_{\mathsf{key}}, k_{\mathrm{F}}, \mathsf{crs}))$
    **assert** $\Sigma.\mathsf{Vrfy}(\mathsf{vk}_{\mathsf{att}}, \mathrm{m}, \sigma_{\mathsf{DE}})$
    $\mathsf{sk} = \mathsf{PKE.Dec}(\mathsf{sk}_{\mathsf{DF}}, \mathsf{ct}_{\mathsf{key}})$
    **assert** $\mathsf{N}.\mathcal{V}((\mathsf{mpk}, \mathsf{ct}), \pi, \mathsf{crs})$
    $(\mathrm{x}, k) = \mathsf{PKE.Dec}(\mathsf{sk}, \mathsf{ct})$
    **assert** $k_{\mathrm{F}} \geq k$
    $(\mathsf{out}, \mathsf{mem}') = \mathrm{F}(\mathrm{x}, \mathsf{mem})$
    **store** $\mathsf{mem} \leftarrow \mathsf{mem}'$
    **return** $(\mathrm{computed}, \mathsf{out})$
```

5.2.3 Proof of security

We now formally state the security guarantees of DoubleSteel.

Theorem 5.1. *For a class of functions* $\mathtt{F}$*, CCA-secure encryption scheme* PKE*, EU-CMA secure signature scheme* Σ*, and non-interactive zero-knowledge proof system* N*, Protocol* DoubleSteel$[\mathtt{F}, \mathsf{PKE}, \Sigma, \mathsf{N}, \lambda, \mathcal{P}, \mathsf{C}]$ *UC-realises ideal functionality DD-FESR*$[\mathtt{F}, \mathcal{P}]$*, in the presence of global functionality* $\mathcal{G}_{\mathsf{att}}$.

Proof. We first construct a simulator, $\mathcal{S}_{\text{DD-FESR}}$. For simplicity of exposition, we use the simulator $\mathcal{S}_{\text{FESR}}$ from Section 3.4 as a subroutine to $\mathcal{S}_{\text{DD-FESR}}$. We instantiate $\mathcal{S}_{\text{FESR}}$ such that shared functionalities (e.g., the secure channels between multiple parties) are implemented by $\mathcal{S}_{\text{DD-FESR}}$, so that it can intercept messages to the parties whose behaviour is simulating and act accordingly. $\mathcal{S}_{\text{DD-FESR}}$ also acts as the ideal functionality in the eyes of the $\mathcal{S}_{\text{FESR}}$ simulator.

By updating the set $\mathbf{B}$ (initialised as empty), $\mathcal{S}_{\text{FESR}}$ keeps record of the parties that are set up as decryptors. We assume that at least one party in the set $\mathbf{B} \cup \{\mathsf{C}\}$ is corrupted at the start of the protocol. We choose this party, GG, to install all G_{att} enclaves for all participants in the protocol, be they honest or corrupted. Due to the property of anonymous attestation guaranteed by G_{att}, the simulator can install all programs on the same machine to produce the attested trace of the real-world protocol, as long as it does not allow GG to execute other parties' enclaves on its own initiative (we use the table $\mathcal{G}$ from simulator $\mathcal{S}_{\text{FESR}}$ to keep track of which party installed each enclave). Similar to the original simulator, we use the shorthand "output $\leftarrow$ $G_{\mathsf{att}}.\mathsf{command}(\mathsf{input})$" to indicate "**simulate sending** (COMMAND, input) **to** G_{att} **through** GG and **receive** output"; note, in Section 3.4, the message was always sent from B instead, given the simpler setting of one honest C and one corrupted B in their proof.

Below, we reproduce the original $\mathcal{S}_{\text{FESR}}$, with appropriate modifications to conform to the syntax of DD-FESR and DoubleSteel.

Simulator $\mathcal{S}_{\text{FESR}}$

State variables	Description
$\mathcal{H}[\cdot] \leftarrow \emptyset$	Table of ciphertext and handles in public repository
$\mathcal{K} \leftarrow []$	List of $\mathsf{prog}_{\mathsf{FE}^{\mathsf{vk}}[\mathsf{F}]}$ enclaves and their $\mathsf{eid}_{\mathsf{F}}$
$\mathbf{B} \leftarrow \{\}$	Set of parties set up as decryptors
$\mathcal{G} \leftarrow \{\}$	Collects all messages sent to G_{att} and its response
$\mathcal{B} \leftarrow \{\}$	Collects all messages signed by G_{att}
$(\mathsf{crs}, \tau) \leftarrow \mathsf{N}.\mathcal{S}_1$	Simulated reference string and trapdoor

```
On message (SETUP, role, $P$) from $\mathcal{S}_{DD\text{-}FESR}$:
  if $\mathsf{mpk} = \perp$ then
    $\mathsf{eid}_{\mathsf{KME}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{C.sid}, \mathsf{prog}_{\mathsf{KME}^{\mathsf{vk}}})$
    $(\mathsf{mpk}, \sigma_{\mathsf{KME}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, \mathrm{init}, \mathsf{crs}, \mathsf{C.sid})$
  if $\mathsf{role} = \mathsf{encryptor}$ then
    send (SETUP, $\mathsf{mpk}, \sigma_{\mathsf{KME}}$) to $\mathcal{SC}_{\mathsf{C}\to P}$
  else if $\mathsf{role} = \mathsf{decryptor}$ then
    send (SETUP, $\mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}}$) to $\mathcal{SC}_{\mathsf{C}\to P}$ and receive (PROVISION, $\sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}$)
    assert $(\mathsf{C.sid}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{vk}}}, \mathsf{pk}_{KD}) \in \mathcal{B}[\sigma]$
    $(\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathrm{provision}, (\sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs})))$
    send (PROVISION, $\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}$) to $\mathcal{SC}_{\mathsf{C}\to P}$
    $\mathbf{B} \leftarrow \mathbf{B} \cup \{P\}$

On message (READ, h) from party B to $\mathcal{REP}$:
    send $(\text{DECRYPT}, \mathsf{F}_0, \mathsf{h})$ to $\mathcal{S}_{\text{DD-FESR}}$ on behalf of B and receive $(\text{DECRYPTED}, |(\mathsf{m}, k)|)$
    assert $|(\mathsf{m}, k)| \neq \perp$
    $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{mpk}, 0^{|(\mathsf{m},k)|})$
    $\pi \leftarrow \mathsf{N}.\mathcal{S}_2(\mathsf{crs}, \tau, (\mathsf{mpk}, \mathsf{ct}))$
    $\mathsf{ct}_{\mathsf{msg}} \leftarrow (\mathsf{ct}, \pi)$; $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] \leftarrow h$
    send $(\text{READ}, \mathsf{ct}_{\mathsf{msg}})$ to B
On message $(\text{INSTALL}, \mathsf{idx}, \mathsf{prog})$ from party $P \in \mathbf{B} \cup \{\mathsf{C}\}$ to $G_{\mathsf{att}}$:
    $\mathsf{eid} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{idx}, \mathsf{prog})$
    $\mathcal{G}[\mathsf{eid}].\mathsf{install} \leftarrow (\mathsf{idx}, \mathsf{prog}, P)$        ▷ $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1]$ is the program's code
    forward eid to B
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input})$ from party $P \in \mathbf{B} \cup \{\mathsf{C}\}$ to $G_{\mathsf{att}}$:
    assert $\mathcal{G}[\mathsf{eid}].\mathsf{install}[2] = P$
    if $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] \neq \mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\cdot]} \vee (\mathsf{input}[0] \neq \mathsf{run} \vee \mathsf{input}[-1] \neq \perp)$ then
        $(\mathsf{output}, \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}, \mathsf{input})$
        $\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \,||\, (\sigma, \mathsf{input}, \mathsf{output})$
        $\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], \mathsf{output})$
        forward $(\mathsf{output}, \sigma)$ to $P$
    else
        $(\mathsf{idx}, \mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]}, P) \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{install}$
        $(\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, k_{\mathrm{F}}, \mathsf{crs}, \perp) \leftarrow \mathsf{input}$
        assert $(\sigma_{\mathrm{F}}, (\mathrm{init}, \mathsf{mpk}, \mathsf{idx}), (\mathsf{pk}_{\mathsf{DF}})) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
        assert $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}_{\mathsf{FE}^{\mathsf{VK}}[\mathrm{F}]}, \mathsf{pk}_{\mathsf{DF}}) \in \mathcal{B}[\sigma_{\mathrm{F}}]$
        assert $(\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}, \mathsf{ct}_{\mathsf{key}}, k_{\mathrm{F}}, \mathsf{crs})) \in \mathcal{B}[\sigma_{\mathsf{DE}}]$
        if $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] = \perp$ then        ▷ If the ciphertext was not computed honestly and saved to $\mathcal{H}$
            $(\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
            $((\mathsf{m}, k), \mathrm{r}) \leftarrow \mathsf{N}.\mathcal{E}(\tau, (\mathsf{mpk}, \mathsf{ct}), \pi)$
            send $(\text{ENCRYPT}, \mathsf{m}, k)$ to $\mathcal{S}_{\text{DD-FESR}}$ on behalf of $P$ and receive $(\text{ENCRYPTED}, \mathsf{h})$
            if $\mathsf{h} \neq \perp$ then $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] \leftarrow \mathsf{h}$
            else return
        $\mathsf{h} \leftarrow \mathcal{H}[\mathsf{ct}_{\mathsf{msg}}]$
        send $(\text{DECRYPT}, \mathrm{F}, \mathsf{h})$ to $\mathcal{S}_{\text{DD-FESR}}$ on behalf of $P$ and receive $(\text{DECRYPTED}, \mathrm{y})$
        $((\mathrm{computed}, \mathrm{y}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \perp, \perp, \perp, \perp, \perp, \perp, \mathrm{y}))$
        $\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \,||\, (\sigma, \mathsf{input}, (\mathrm{computed}, \mathrm{y})))$
        $\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], (\mathrm{computed}, \mathrm{y}))$
        forward $((\mathrm{computed}, \mathrm{y}), \sigma)$ to $P$
```

We give $\mathcal{S}_{\text{DD-FESR}}$ white-box access to $\mathcal{S}_{\text{FESR}}$, letting the former freely access the internal tapes of the latter. We mark the names of the variables that $\mathcal{S}_{\text{DD-FESR}}$ from $\mathcal{S}_{\text{FESR}}$ in the state variable declaration below. In particular, we use the internal values of $\mathcal{S}_{\text{FESR}}$ to keep track of messages sent to the enclaves and their attestation signatures. For all calls to an enclave, the $\mathcal{S}_{\text{DD-FESR}}$ simulator always activates $\mathcal{S}_{\text{FESR}}$ so that these internal variables can be updated. The simulator queries the environment for additional inputs by activating the dummy adversary via messages sent to the corrupted parties.

Simulator $\mathcal{S}_{\text{DD}-\text{FESR}}$

State variables	Description
$K \leftarrow \{\}$	set of **A** keypairs and $\mathcal{F}_{\mathsf{CERT}}$ certificates
$KS \leftarrow \{\}$	set of generated keyshares
$\mathsf{GG} \leftarrow \bot$	The corrupted party on which we run simulated enclaves
$\mathcal{G} = \mathcal{S}_{\text{FESR}}.\mathcal{G}$	Collects all messages sent to G_{att} and its response
$\mathcal{B} = \mathcal{S}_{\text{FESR}}.\mathcal{B}$	Collects all messages signed by G_{att}
$\mathsf{crs} = \mathcal{S}_{\text{FESR}}.\mathsf{crs}$	Simulated common reference string

```
On message $(\textsc{Setup}, \mathsf{role}, P)$ from DD-FESR:
  if $\mathsf{role} = \mathsf{encryptor}$ then
    if $P$ is honest then
      if $\mathsf{C}$ is honest then
        send $(\textsc{setup}, \mathsf{encryptor}, P)$ to $\mathcal{S}_{\text{FESR}}$
      else
        send $(\textsc{Setup}, \mathsf{encryptor})$ to $\mathcal{SC}_{P \to \mathsf{C}}$ and receive $\mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}}$
        assert $(\mathsf{C.sid}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{prog}_{\mathsf{KME}^{\mathsf{VK}}}, \mathsf{mpk}) \in \mathcal{B}[\sigma_{\mathsf{KME}}]$
      $\mathsf{vk}_{\mathsf{att}} \leftarrow G_{\mathsf{att}}.\mathsf{getPK}()$
      send $\textsc{GetK}$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive $\mathsf{VK}$
      $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$
      simulate sending $(\textsc{Sign}, \mathsf{spk})$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ through P and receive $\mathsf{cert}$
      $K[P] \leftarrow (\mathsf{spk}, \mathsf{ssk}, \mathsf{cert})$
      send OK to DD-FESR
    else
      if $\mathsf{C}$ is honest then
        simulate sending $(\textsc{Setup}, \mathsf{encryptor})$ to $P$ on behalf of $\mathcal{Z}$
        await for message $(\textsc{setup}, \mathsf{encryptor}, P)$ on $\mathcal{SC}_{P \to \mathsf{C}}$
        send $(\textsc{setup}, \mathsf{encryptor}, P)$ to $\mathcal{S}_{\text{FESR}}$
        send OK to DD-FESR
```

```
            else
                simulate sending $(\textsc{setup}, \mathsf{encryptor})$ to $P$ on behalf of $\mathcal{Z}$
                await for message $(\textsc{Sign}, \mathsf{spk})$ from $P$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$
                send OK to DD-FESR
    else if $\mathsf{role} = \mathsf{decryptor}$ then
        if $P$ is honest then
            if $\mathsf{C}$ is honest then
                notify $\mathcal{S}_{\text{FESR}}$ that $(\textsc{install}, \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}})$ was sent from $P$ to $G_{\mathsf{att}}$ and capture
    response $\mathsf{eid}_{\mathsf{DE}}$
                send $(\textsc{setup}, \mathsf{decryptor}, P)$ to $\mathcal{S}_{\text{FESR}}$ and receive $\mathsf{mpk}, \sigma_{\mathsf{KME}}, \mathsf{eid}_{\mathsf{KME}}$
                notify $\mathcal{S}_{\text{FESR}}$ that $(\textsc{resume}, \text{init-setup}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}, P.\mathsf{sid})$ was sent from $P$ to
    $G_{\mathsf{att}}$ and capture response $(\mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}), \sigma_{\mathsf{init}}$
                send $(\textsc{provision}, \sigma_{\mathsf{init}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$ to $\mathcal{S}_{\text{FESR}}$ and receive $(\textsc{provision}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}})$
                notify $\mathcal{S}_{\text{FESR}}$ that $(\textsc{resume}, \text{complete-setup}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}})$ was sent from $P$ to $G_{\mathsf{att}}$
                send OK to DD-FESR
            else
                simulate sending $(\textsc{setup}, \mathsf{decryptor})$ to $P$ on behalf of $\mathcal{Z}$
                await for $(\textsc{setup}, \mathsf{decryptor})$ message on $\mathcal{SC}_{\mathsf{C} \to P}$
                send $(\textsc{setup}, \mathsf{decryptor}, P)$ to $\mathcal{S}_{\text{FESR}}$
        else
            if $\mathsf{C}$ is honest then
                send $(\textsc{setup}, \mathsf{decryptor}, P)$ to $\mathcal{S}_{\text{FESR}}$
                send OK to DD-FESR
            else
                simulate sending $(\textsc{setup}, \mathsf{decryptor})$ to $P$ on behalf of $\mathcal{Z}$
                await for $(\textsc{resume}, \text{complete-setup}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}})$ from $P$ to $G_{\mathsf{att}}$
                $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{ct}_{\mathsf{key}}) \leftarrow \mathcal{B}[\sigma_{\mathsf{KME}}]$
                assert $\mathsf{idx} = P.\mathsf{sid} \land \mathsf{prog} = \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}} \land (\sigma_{\mathsf{KME}}, \cdot, \mathsf{ct}_{\mathsf{key}}) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
                send OK to DD-FESR
On message $(\textsc{Sign}, \mathsf{spk})$ from corrupted party $P$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$:
    forward $(\textsc{Sign}, \mathsf{spk})$ to $\mathcal{F}_{\mathsf{CERT}}(\mathcal{P})$ and receive response cert
    $K[P] \leftarrow (\mathsf{spk}, \bot, \mathsf{cert})$
    return cert
On message $(\textsc{keysharequery}, x, \mathsf{A}, \mathsf{B})$ from DD-FESR:
    // A is corrupted
    send $(\textsc{KeyShareGen}, x, \mathsf{A}, \mathsf{B})$ to $\mathsf{A}$ on behalf of $\mathcal{Z}$ and await for $(\textsc{KeyShareGen},$
    $x, \mathsf{A}, \mathsf{B})$ from A to $\mathcal{SC}_{\mathsf{A}\to\mathsf{B}}$
    **return** OK
On message (KEYSHAREGEN, $f, \mathsf{A}, \mathsf{B}$) from DD-FESR:
    **if** A is honest **then**
        **if** B is honest **then** $\mathrm{F} \leftarrow \mathrm{F}_0$
        **else** $\mathrm{F} \leftarrow f$
        $(\mathsf{spk}, \mathsf{ssk}, \mathsf{cert}) \leftarrow K[\mathsf{A}]$
        $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{ssk}, \mathrm{F}, \mathsf{B})$
        $KS[\mathrm{F}, \mathsf{A}, \mathsf{B}] \leftarrow \sigma$
        **send** (KEYSHAREGEN, $\mathrm{F}, \sigma, \mathsf{spk}, \mathsf{cert}$) **to** $\mathcal{SC}_{\mathsf{A}\to\mathsf{B}}$
On message (RESUME, eid, input) from corrupted party P to $G_{\mathsf{att}}$:
    **if** $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] = \mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}} \wedge \mathsf{input}[0] = $ provision **then**
        $(\text{provision}, \mathcal{KS}_{\mathrm{F}}, \mathsf{eid}_{\mathrm{F}}, \mathsf{pk}_{\mathsf{DF}}, \sigma_{\mathrm{F}}, \mathrm{F}, \mathsf{pid}) \leftarrow \mathsf{input}$
        **for** $(\sigma, \mathsf{spk}, \mathsf{cert}) \in \mathcal{KS}_{\mathrm{F}}$ **do**
            **assert** $(\mathsf{spk}, \cdot, \mathsf{cert}) = K[\mathsf{A}]$ for some A
            **if** $\sigma \notin KS[\mathrm{F}, \mathsf{A}, P]$ **then**
                $KS[\mathrm{F}, \mathsf{A}, P] \leftarrow \sigma$
                **send** KeyShareGen **to** $\mathrm{F}, P$ **through** A and **receive** DD-FESR
    (KEYSHAREQUERY, $\mathrm{F}, \mathsf{A}, P$)
                **send** OK **to** DD-FESR
    **send** (RESUME, eid, input) **to** $\mathcal{S}_{\mathrm{FESR}}$
On message (ENCRYPT, input) from $\mathcal{S}_{\mathrm{FESR}}$ on behalf of P:
    **send** (ENCRYPT, input) **to** DD-FESR **through** $P$ and **receive** (ENCRYPTED, output)
    **send** (ENCRYPTED, output) **to** $\mathcal{S}_{\mathrm{FESR}}$ **on behalf of** DD-FESR
On message (DECRYPT, input) from $\mathcal{S}_{\mathrm{FESR}}$ on behalf of P:
    **send** (DECRYPT, input) **to** DD-FESR **through** $P$ and **receive**
    (DECRYPTED, output)
    **send** (DECRYPTED, output) **to** $\mathcal{S}_{\mathrm{FESR}}$ **on behalf of** DD-FESR
On message *:
    forward * to $\mathcal{S}_{\mathrm{FESR}}$
```

We now show, via a series of hybrid experiments, that given the above simulator, the real and ideal worlds are indistinguishable from the environment's viewpoint. We begin with the real-world protocol, which can be considered as *Hybrid* 0.

Hybrid 1 consists of the ideal protocol for DD-FESR, which includes the relevant dummy parties, and the simulator $\mathcal{S}'_{\text{DD-FESR}}$, which on any message from the environment ignores the output of the ideal functionality, and faithfully reproduces protocol DoubleSteel. The equivalence between *Hybrid*s 0 and 1 is trivial due to the behaviour of $\mathcal{S}'_{\text{DD-FESR}}$.

Hybrid 2 replaces all operations of $\mathcal{S}'_{\text{DD-FESR}}$ where the protocol DoubleSteel behaves in the same way as Steel (except that it sends messages with the full set of arguments expected by DoubleSteel rather than those in Steel, and receives the equivalent DoubleSteel return values) with a call to an emulated $\mathcal{S}_{\text{FESR}}$. Due to the security proof of the Steel protocol in Section 3.4, we now use the simulator of *Hybrid* 2 to simulate FESR with respect to protocol Steel, making the two hybrids indistinguishable. An environment that is able to distinguish between the two hybrids could create an adversary that can distinguish between executions of FESR and Steel; but due to the UC emulation statement, no such environment can exist in the presence of $\mathcal{S}_{\text{FESR}}$. The reduction to $\mathcal{S}_{\text{FESR}}$ greatly simplifies the current proof, as we are guaranteed the security of the secure key provisioning and decryption due to the similarities of these two phases of the protocols between Steel and DoubleSteel.

Hybrid 3 modifies the simulator of *Hybrid* 2 by replacing all the signature verification operations for attestation signatures in DD-FESR with a table lookup from $\mathcal{S}_{\text{FESR}}.\mathcal{B}$. The new table lookups for attestation signatures complement the ones enacted by $\mathcal{S}_{\text{FESR}}$, while capturing behaviour that is unique to DoubleSteel. Similar to Lemma 3.2, if the environment can distinguish between this hybrid and the previous one, it can construct an adversary to break the unforgeability of signatures.

Hybrid 4 modifies the simulator of *Hybrid* 3 by replacing KEYSHAREGEN and KEYSHAREQUERY requests for any functions with a request for a dummy function (such as the natural leakage function F_0), for all these requests where both the encryptor and the decryptor are honest. The environment is not able to distinguish between the two hybrids due to the security of the secure channel functionality (as defined in Section 2.3.6): the secure channel only leaks the length of a message exchanged between sender and receiver, and assuming that we represent functions with a fixed-length string (such as a hash of its code), the leakage between this hybrid and the previous one is indistinguishable.

Hybrid 5 adds an additional check to the simulator of *Hybrid* 4 before it can run the provision command on enclave $\mathsf{prog}_{\mathsf{DE}^{\mathsf{VK}}}$ through the internal $\mathcal{S}_{\text{FESR}}$ simulator. The check ensures that all keyshares passed by the malicious decryptor to the enclave are signed by a party who has first registered their verification key with the certificate authority. Then, if the signature has not been generated through a call to the ideal functionality but rather through a local signing operation, the simulator notifies the ideal functionality to update its internal keyshare count. This hybrid essentially replaces the algorithmic signature verification operations in the previous one with two table lookups (for both verification key certification and keyshare authenticity). If an adversary was able to bypass the checks by providing either a certificate that wasn't produced by the ideal functionality or a keyshare that didn't match with the triple of $(\mathsf{F}, \mathsf{A}, \mathsf{B})$, they would be able to create an adversary that could break the unforgeability of signature scheme Σ in the same manner as in Hybrid 3. Thus the hybrid is indistinguishable from Hybrid 4.

The simulator defined in *Hybrid* 5 is identical to $\mathcal{S}_{\text{DD-FESR}}$; thus, it holds that DoubleSteel UC-emulates DD-FESR. □

5.2.4 Making Multi-Input functions Stateful

As pointed out in Chapter 3, FESR subsumes Multi-Input Functional Encryption [144] in that it is possible to use the state to emulate functions over multiple inputs. We now briefly outline how to realise a Multi-Input functionality using FESR (and by extension DD-FESR) through the definition of a simple compiler from multi-input functions to single-input stateful functions. To compute a stateless, multi-input function $\mathrm{F} : (\underbrace{\mathcal{X} \times \cdots \times \mathcal{X}}_{n}) \to \mathcal{Y}$ we define the following stateful functionality:

```
function $\mathsf{Agg}_{\mathrm{F},n}(\mathrm{x}, \mathsf{s})$
    if $|\mathsf{s}| < n$ then return $(\perp, \mathsf{s} \parallel \mathrm{x})$
    else return $(\mathrm{F}(\mathsf{s}||\mathrm{x}), \emptyset)$        $\triangleright$ s is equivalent to the array containing $\mathrm{x}_1, \ldots, \mathrm{x}_{n-1}$
```

where input x is in $\mathcal{X}$, the state s is in $\mathcal{S}$, and n is bounded by the maximum size of $\mathcal{S}$. The above aggregator function is able to merge the inputs of multiple encryptors because in FESR the state of a function is distinct between each decryptor; therefore, multiple decryptors attempting to aggregate inputs will not interfere with each other's functions. There are several possible extensions to the above compiler:

1. In $\mathsf{Agg}_{\mathrm{F},n}(\cdot)$, the order of parameters relies on the decryptor's sequence of invocations. If F is a function where the order of inputs affects the result, malicious decryptor B could choose not to run decryption in the same order of inputs as received. It is possible to further extend the decryption function to respect the order of parameters set by each encryptor. If a subset of encryptors is malicious, we can parameterise the function by a set of public keys for each party, and ask them to sign their inputs.

2. The compiler can be easily extended to multi-input *stateful* functionalities, by keeping a list of inputs (as a field) within the state array and not discarding the state on the n-th invocation of the compiler.

3. One additional advantage of implementing Multiple-Input functionalities through stateful functionalities is that we are not constrained to functions with a fixed number of inputs. If we treat the inner functionality to the compiler as a function taking as input a list, we can use the same compiler functionality for inner functions of any n-arity (we denote this type of functions as $[\![\mathcal{X}]\!] \to \mathcal{Y}$). On the first (integer) input to the aggregator, we set it as a special field n in the state, and for the next $n-1$ calls we simply append the inputs to the state, while returning the empty value. On the nth call, we execute the function on the stored state field (now containing n entries), erase the state and index from memory and wait for the next call to set a new value to n.

In the next section, we will use the compiler AggS_F, which combines the above defined compiler extensions 2 and 3 to compute any stateful function F with variable number of inputs ($\mathsf{F} : [\![\mathcal{X}]\!] \times \mathcal{S} \times \mathcal{R} \to \mathcal{Y} \times \mathcal{S}$) from DD-FESR.

5.3 The GlassVault protocol

First, we provide the formal definition of "Analysis-augmented Exposure Notification" (EN^+), an extension of the standard Exposure Notification (EN), to allow arbitrary computation on data shared by users. Our definition of EN^+ is built on the Exposure notification functionality $\mathcal{F}_{\mathrm{EN}}$ of Canetti et al. [84]. Then, we present a description of protocol GlassVault, and show it UC-realises the ideal functionality of EN^+, i.e., $\mathcal{F}_{\mathrm{EN}^+}$.

5.3.1 Analysis-augmented Exposure Notification

Since EN^+ is built upon EN, we first re-state relevant notions used in the UC modelling of EN. Specifically, EN relies on the time functionality $\mathbb{T}$ and the *physical reality* functionality $\mathbb{R}$ that we present in Sections 5.1.3.2 and Section 5.1.3.3, respectively. In particular, $\mathbb{R}$ models the occurrence of events in the physical world (e.g., users' motion or location data). Measurements of a real-world event are sent as input from the environment, and each party can retrieve a list of their own measurements. The functionality only accepts new events if they are "physically sensible", and can send the entire list of events to some privileged entities such as ideal functionalities. Using $\mathbb{T}$ and $\mathbb{R}$ as subroutines, the functionality $\mathcal{F}_{\text{EN}}$ (presented formally in Section 5.1.3.5), is defined in terms of a risk estimation function ρ, a leakage function $\mathcal{L}$, a set of allowable measurement error functions E, and a set of allowable faking functions Φ. The functionality queries $\mathbb{R}$ and applies the measurement error function chosen from E by the simulator to compute a *noisy record of reality*. This in turn is used to decide whether to mark a user as infected, and to compute a risk estimation score for any user. The adversary can mark some parties as corrupted and obtain leakage of their local state, as well as modifying the physical reality record with a reality-faking function. This allows simulation of adversarial behaviour, such as relay attacks (where an infectious user appears to be within transmission distance from a non-infectious malicious user). The functionality captures a variety of contact tracing protocols and attacker models via its parameters. For simplicity, it does not model the testing process the users engage in to find out they are positive, and it assumes that once a user is notified of exposure they are removed from the protocol.

The extension to EN$^+$ involves an additional type of entity to the above scheme, namely, analyst Ä, who wants to learn a certain function, α, on additional data contributed by users, some of the data might be sensitive (denoted by physical reality field label SEC). Thus, the users are provided with a mechanism to accept whether an analyst is allowed to receive the result of the executions of any particular function. In order to receive the result, the analyst needs to be authorised by a portion of exposed users determined by function K. The choice of this function is a trade-off between liveness and security: a higher threshold might result in fewer functions being authorised, while a lower one might make it easier to authorise undesirable functions by colluding parties.

We now present a formal definition of $\mathcal{F}_{\text{EN}^+}$. Highlighted sections of the functionality represent where EN$^+$ diverges from EN.

Functionality $\mathcal{F}_{\text{EN}^+}[\rho, E, \Phi, \mathcal{L}, AF, K, \mathcal{P}]$

The functionality is parameterised by exposure risk function ρ, a set of allowable error functions E for the physical reality record, a set of faking functions Φ for the adversary to misrepresent the physical reality, and a leakage function $\mathcal{L}$, as in the regular $\mathcal{F}_{\text{EN}}$. AF is the set of all functions $\{\alpha \mid \alpha : [\![\mathcal{X}]\!] \times \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{Y} \times \mathcal{S}\}$ an analyst could be authorised to compute. $K()$ is a function of the current number of users required to determine the minimum threshold of analyst authorisations. $\mathcal{P}$ is a set of (dummy) parties.

State variables	Description
$\mathbf{SE}$	List of users who have shared their exposure status and time of upload
$\overline{\mathbf{U}}$	List of active users; $\mathbf{SE} \cap \overline{\mathbf{U}} = \emptyset$
$\widetilde{\mathbf{U}}$	List of corrupted users
$\overline{A}$	For each pair of analyst and allowed function, the dictionary $\overline{A}$ contains the users that have authorised this pair
$\widetilde{\ddot{\mathrm{A}}}$	Static set of corrupted analysts
$\mathcal{ST}$	State table for (function, analyst) pairs

```
On message $(\text{SETUP}, \varepsilon^*)$ from $\mathcal{A}$:
    assert $\varepsilon^* \in E$
    $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \emptyset$                ▷ Initialise noisy record of physical reality
On message ACTIVATEMOBILEUSER from $U \in \mathcal{P}$:
    $\overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \parallel U$
    send $(\text{ACTIVATEMOBILEUSER}, U)$ to $\mathcal{A}$
On message SHAREEXPOSURE from $U \in \mathcal{P}$:
    send $(\text{ALLMEAS}, \varepsilon^*)$ to $\mathbb{R}$ and receive $\widetilde{\mathrm{R}}^*$
    $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \widetilde{\mathrm{R}}_\varepsilon \parallel \widetilde{\mathrm{R}}^*$
    if $\widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{INFECTED}] = \top$ then
        send TIME to $\mathbb{T}$ and receive $t$
        $\mathbf{SE} \leftarrow \mathbf{SE} \parallel (U,t); \overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \setminus \{U\}$
    if $U \in \widetilde{\mathbf{U}}$ then
        send $(\text{SHAREEXPOSURE}, U, \widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{SEC}], \widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{INFECTED}])$ to $\mathcal{A}$
    if $U \notin \widetilde{\mathbf{U}} \wedge \widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{INFECTED}] = \top$ then
        send $(\text{SHAREEXPOSURE}, U, \perp, \widetilde{\mathrm{R}}_\varepsilon[U][\mathsf{INFECTED}])$ to $\mathcal{A}$
On message EXPOSURECHECK from $U \in \mathcal{P}$:
    if $U \in \overline{\mathbf{U}}$ then
        send $(\text{ALLMEAS}, \varepsilon^*)$ to $\mathbb{R}$ and receive $\widetilde{\mathrm{R}}^*$
        $\widetilde{\mathrm{R}}_\varepsilon \leftarrow \widetilde{\mathrm{R}}_\varepsilon \parallel \widetilde{\mathrm{R}}^*; \mu \leftarrow \widetilde{\mathrm{R}}_\varepsilon[U] \parallel \widetilde{\mathrm{R}}_\varepsilon[\mathbf{SE}]$
        return $\rho(U, \mu)$
    else return error
On message $(\text{REGISTERANALYST}, \alpha)$ from $\ddot{\mathrm{A}} \in \mathcal{P}$:
  if $\alpha \in AF$ then
      send $(\textsc{RegisterAnalyst}, \alpha, \ddot{A})$ to $\mathcal{A}$
      for all $U \in \mathbf{SE} \cup \overline{\mathbf{U}}$ do send $(\textsc{RegisterAnalystRequest}, \alpha, \ddot{A})$ to $U$
      $\overline{A}[\alpha, \ddot{A}] \leftarrow []$
On message $(\textsc{RegisterAnalystAccept}, \alpha, \ddot{A})$ from $U \in \mathcal{P}$:
  if $U \in \widetilde{\mathbf{U}} \vee \ddot{A} \in \widetilde{\ddot{\mathbf{A}}}$ then send $(\textsc{RegisterAnalystAccept}, \alpha, \ddot{A}, U)$ to $\mathcal{A}$ and receive OK
  $\overline{A}[\alpha, \ddot{A}] \leftarrow \overline{A}[\alpha, \ddot{A}] \parallel U$
  send $(\textsc{RegisterAnalystAccept}, U, \alpha)$ to $\ddot{A}$
On message $(\textsc{Analyse}, \alpha)$ from $\ddot{A} \in \mathcal{P}$:
  if $|\overline{A}[\ddot{A}, \alpha]| \geq K(|\mathbf{SE}| + |\overline{\mathbf{U}}|)$ then
      $(y, \mathcal{ST}') \leftarrow \alpha(\widetilde{R}_\varepsilon[\mathbf{SE} \cup \overline{\mathbf{U}}][\mathsf{SEC}], \mathcal{ST}[\alpha, \ddot{A}])$
      $\mathcal{ST}[\alpha, \ddot{A}] \leftarrow \mathcal{ST}'$
      if $\ddot{A} \in \widetilde{\ddot{\mathbf{A}}}$ then
          send $(\textsc{Analysed}, \alpha, \ddot{A})$ to $\mathcal{A}$ and receive $OK$
      return $(\textsc{analysed}, y)$
On message $\textsc{RemoveMobileUser}$ from $U \in \mathcal{P}$:
  $\overline{\mathbf{U}} \leftarrow \overline{\mathbf{U}} \setminus \{U\}$
On message $(\textsc{Corrupt}, U)$ from $\mathcal{A}$:
  $\widetilde{\mathbf{U}} \leftarrow \widetilde{\mathbf{U}} \parallel U$
  return $\{(\alpha, \ddot{A}) : U \in \overline{A}[\alpha, \ddot{A}]\}$
On message $(\textsc{MyCurrentMeas}, U, A, e)$ from $\mathcal{A}$:
  if $U \in \widetilde{\mathbf{U}}$ then
      send $(\textsc{MyCurrentMeas}, U, A, e)$ to $\mathbb{R}$ and receive $u_A^e$
      send $(\textsc{MyCurrentMeas}, u_A^e)$ to $\mathcal{A}$
On message $(\textsc{FakeReality}, \phi)$ from $\mathcal{A}$:
  if $\phi \in \Phi$ then $\widetilde{R}_\varepsilon \leftarrow \phi(\widetilde{R}_\varepsilon)$
On message $\textsc{Leak}$ from $\mathcal{A}$:
  send $(\textsc{Leak}, \mathcal{L}(\{\widetilde{R}_\varepsilon, \overline{\mathbf{U}}, \mathbf{SE}\}))$ to $\mathcal{A}$
On message $(\textsc{IsCorrupt}, U)$ from $\mathcal{Z}$:
  return $U \overset{?}{\in} \widetilde{\mathbf{U}}$
```

5.3.2 GlassVault Protocol

In this section, we present the GlassVault protocol. It is a delicate combination of two primary primitives; namely, (i) the original exposure notification proposed by Canetti et al. [84], and (ii) the enhanced functional encryption scheme that we proposed in Section 5.2. In this protocol, the users upload not only the regular data needed for exposure notification but also the encryption of their sensitive measurements (e.g., their GPS coordinates, electronic health records, environment's air quality). Once an analyst requests to execute some specific computations on the users' data, the users are informed via public announcements. At this stage, users can provide permission tokens to the analyst (in the form of functional key shares), who can run such computations only if the number of tokens exceeds a threshold defined by the function K over the number of parties in the set $\mathbf{A}$ of DD-FESR. Due to the security of the proposed functional encryption scheme, the GlassVault analyst does not learn anything about the users' sensitive inputs beyond what the function evaluation reveals.

The protocol supports our goals of providing a generic and accountable analysis service. GlassVault is generic, as it supports arbitrary secure computations (i.e., multi-input stateful and randomised functions) on users' shared data. It is also accountable, as computations are performed only if permission is granted by a sufficient number of users and in that the user can choose whether they are willing to share their sensitive data or not, making the collection of information consensual. Besides DD-FESR functionality, the GlassVault protocol makes use of the physical reality functionality $\mathbb{R}$, the exposure notification functionality $\mathcal{F}_{\text{EN}}$, and the trusted bulletin board functionality $\mathcal{F}_{\text{TBB}}$, described in Sections 5.1.3.3, 5.1.3.5, and 5.1.3.4, respectively.

Protocol GlassVault$[\rho, E, \Phi, \mathcal{L}, AF, K, \mathcal{P}]$

The protocol takes the same class of parameters as defined in $\mathcal{F}_{\text{EN}^+}$. We use U to refer to a normal user of the Exposure Notification System (corresponding to A in DD-FESR). We use Ä to refer to an analyst (corresponding to DD-FESR's decryptor, B). Among other ideal setups, GlassVault leverages the exposure notification ideal functionality EN$[\rho, E, \Phi, \mathcal{L}, \mathcal{P}]$ and functional encryption ideal functionality DD-FESR$[AF, \mathcal{P}]$.

User $U \in \mathcal{P}$:

On message ACTIVATEMOBILEUSER *from* $\mathcal{Z}$:

    send ACTIVATEMOBILEUSER **to** $\mathcal{F}_{\text{EN}}$

On message REMOVEMOBILEUSER *from* $\mathcal{Z}$:

```
  send RemoveMobileUser to $\mathcal{F}_{\mathrm{EN}}$
On message ShareExposure from $\mathcal{Z}$:
  send ShareExposure to $\mathcal{F}_{\mathrm{EN}}$ and receive $r$
  send (Setup, encryptor) to DD-FESR
  send (MyCurrentMeas, SEC, $e$) to $\mathbb{R}$ and receive $u^e_{\mathsf{SEC}}$
  send (Encrypt, $u^e_{\mathsf{SEC}}$, $K(|\text{DD-FESR.}\mathbf{A}|)$) to DD-FESR and receive (encrypted, h)
  erase $u^e_{\mathsf{SEC}}$ and send (Add, h) to $\mathcal{F}_{\mathrm{TBB}}$
On message ExposureCheck from $\mathcal{Z}$:
  send ExposureCheck to $\mathcal{F}_{\mathrm{EN}}$ and receive $\rho_U$
  return $\rho_U$
On message (RegisterAnalystRequest, $\alpha$) from Ä:
  send (RegisterAnalystRequest, $\alpha$, Ä) to $\mathcal{Z}$
On message (RegisterAnalystAccept, $\alpha$, Ä) from $\mathcal{Z}$:
  send (KeyShareGen, $\mathsf{AggS}_\alpha$, Ä) to DD-FESR
  send (RegisterAnalystAccept, $U$, $\alpha$) to Ä
Analyst Ä $\in \mathcal{P}$:
On message (RegisterAnalyst, $\alpha$) from $\mathcal{Z}$:
  send (Setup, decryptor) to DD-FESR
  for all $U \in \mathbf{P} \setminus \{\text{Ä}\}$ do send (RegisterAnalystRequest, $\alpha$) to $U$
On message (Analyse, $\alpha$) from $\mathcal{Z}$:
  send Retrieve to $\mathcal{F}_{\mathrm{TBB}}$ and receive $\mathcal{C}$
  send (Encrypt, $(|\mathcal{C}|, 0)$) to DD-FESR and receive $\mathsf{h}_n$
  send (Decrypt, $\mathsf{h}_n$, $\mathsf{AggS}_\alpha$) to DD-FESR and receive
  (decrypted, $|\mathcal{C}|$)
  for $\mathsf{h} \in \mathcal{C}$ do
    send (Decrypt, h, $\mathsf{AggS}_\alpha$) to DD-FESR and receive (decrypted, y)
    if $\mathsf{y} \neq \perp$ then return (analysed, y)
```

5.3.3 Proof of security

We now show that GlassVault is secure.

Theorem 5.2. *Let* $\rho, E, \Phi, \Phi^+, \mathcal{L}, \mathcal{L}^+, AF, K$, *and* C *be parameters such that the following conditions hold:*

1. $\Phi \subset \Phi^+$,

2. *for every input x, it holds that*[2] $\mathcal{L}(x) = \mathcal{L}(\mathcal{L}^+(x))$*, and*

3. *there is a function* $\phi^+ \in \Phi^+$ *such that for every input x, every noisy physical reality record* $\widetilde{\mathsf{R}}_\varepsilon \in x$*, every function* $\phi \in \Phi$*, and every set of users U,*

   $\phi(\widetilde{\mathsf{R}}_\varepsilon)$ *does not modify the sensitive data stored in record* $\widetilde{\mathsf{R}}_\varepsilon[U][\mathsf{SEC}]$*, but* $\phi^+(\widetilde{\mathsf{R}}_\varepsilon)$ *does, and*

   $\mathcal{L}(x)$ *does not contain any instruction to leak the contents of* $\widetilde{\mathsf{R}}_\varepsilon[U][\mathsf{SEC}]$*, but* $\mathcal{L}^+(x)$ *does.*

Then, it holds that $\mathsf{GlassVault}[\rho, E, \Phi, \mathcal{L}, AF, K, \mathcal{P}]$ *UC-realises* $\mathcal{F}_{\mathsf{EN}^+}[\rho, E, \Phi^+, \mathcal{L}^+, AF, K, \mathcal{P}]$*, in the presence of global functionalities* $\mathbb{T}$ *and* $\mathbb{R}$*.*

Proof. We first give a detailed proof for semi-honest adversaries and sketch how this proof can be adapted for fully corrupted adversaries that diverge from the protocol. We first construct a simulator $\mathcal{S}_{\mathsf{GV}}$.The high-level task of our simulator is to *synchronize* the inputs of the analysis functions between the ideal world (where they are stored in $\mathbb{R}$), and the real world (where they are held in the DD-FESR ideal repository). The simulator updates a simulated trusted bulletin board by obtaining, through the leakage function, the secret data for all honest users who have shared their exposure, and encrypting it through DD-FESR.

When any registered and corrupted analyst executes an ANALYSE request in the ideal world, the simulator allows the ideal functionality to return the ideal result of the computation only if the adversary instructs the analyst to correctly aggregate the ciphertexts stored in the bulletin board through DD-FESR decryption requests to the appropriate aggregator function. We also simulate the REGISTERANALYST and REGISTERANALYSTACCEPT sequence of operations by triggering the corresponding SETUP and KEYSHAREGEN subroutines in DD-FESR. Any other adversarial calls to $\mathcal{F}_{\mathsf{EN}^+}$ such as $(\textsc{Setup}, \varepsilon^*)$ and $(\textsc{FakeReality}, \phi)$ are allowed and redirected to $\mathcal{F}_{\mathsf{EN}}$, as long as $\varepsilon^* \in E$ and $\phi \in \Phi$).

Simulator $\mathcal{S}_{\mathsf{GV}}$

[2]Note, if x is a labelled dictionary and $\mathcal{L}$ returns a dictionary which includes a subset of entries in x and optionally any other additional records, $\mathcal{L}^+$ strictly returns more records than $\mathcal{L}$.

State variables	Description
$\mathcal{L}^+$	function to leak anything that $\mathcal{L}$ does, as well as the contents of SEC for all users
$\mathcal{T} \leftarrow \{\}$	Table that stores messages uploaded to the Trusted Bulletin Board
$\widetilde{\mathbf{U}}$	List of corrupted users
SE	List of exposed users

```
On message $(\textsc{ActivateMobileUser}, U)$ from $\mathcal{F}_{\text{EN}^+}$:
    simulate sending $\textsc{ActivateMobileUser}$ to $\mathcal{F}_{\text{EN}}$ on behalf of $U$
On message $(\textsc{ShareExposure}, U, u^e_{\mathsf{SEC}}, v)$ from $\mathcal{F}_{\text{EN}^+}$:
    simulate sending $(\textsc{Setup}, \mathsf{encryptor})$ to DD-FESR on behalf of $U$
    if $u^e_{\mathsf{SEC}} = \bot$ then
        // simulate honest user:
        send $\textsc{Leak}$ to $\mathcal{F}_{\text{EN}^+}$ and receive $\mathsf{r}$
        $u^e_{\mathsf{SEC}} \leftarrow \mathsf{r}[U][\mathsf{SEC}]$
    simulate sending $\big(\textsc{Encrypt}, u^e_{\mathsf{SEC}}, K(|\text{DD-FESR}.\mathbf{A}|)\big)$ to DD-FESR through $U$ and
    receive $(\textsc{encrypted}, \mathsf{h})$
    $\mathcal{T} \leftarrow \mathcal{T} \,\|\, \mathsf{h}$
    if $v = \top$ then $\mathbf{SE} \leftarrow \mathbf{SE} \,\|\, U$
On message $(\textsc{RegisterAnalyst}, \alpha, \mathsf{Ä})$ from $\mathcal{F}_{\text{EN}^+}$:
    simulate sending $(\textsc{setup}, \mathsf{decryptor})$ to DD-FESR on behalf of $\mathsf{Ä}$
    for $U \in \mathbf{P} \setminus \{\mathsf{Ä}\}$ do
        simulate sending $(\textsc{RegisterAnalystRequest}, \alpha)$ to $U$ on behalf of $\mathsf{Ä}$
On message $(\textsc{RegisterAnalystAccept}, \alpha, \mathsf{Ä}, U)$ from $\mathcal{F}_{\text{EN}^+}$:
    if $U \in \widetilde{\mathbf{U}}$ then
        await for $\big(\textsc{KeyShareGen}, \mathsf{AggS}_\alpha, \mathsf{Ä}\big)$ from $U$ to DD-FESR
    else
        simulate sending $\big(\textsc{KeyShareGen}, \mathsf{AggS}_\alpha, \mathsf{Ä}\big)$ to DD-FESR on behalf of $U$
    return OK
On message $(\textsc{Analysed}, \alpha, \mathsf{Ä})$ from $\mathcal{F}_{\text{EN}^+}$:
    await for $(\textsc{Encrypt}, (|\mathcal{T}|, 0))$ from $\mathsf{Ä}$ to DD-FESR and for response $(\textsc{encrypted}, \mathsf{h}_n)$
    await for $(\textsc{Decrypt}, \mathsf{h}_n, \mathsf{AggS}_\alpha)$ from $\mathsf{Ä}$ to DD-FESR and for response $(\textsc{decrypted}, |\mathcal{T}|)$
    for $\mathsf{h} \in \mathcal{T}$ do
        await for $(\textsc{Decrypt}, \mathsf{h}, \mathsf{AggS}_\alpha)$ from $\mathsf{Ä}$ to DD-FESR and for response $(\textsc{decrypted}, \cdot)$
    return OK
On message LEAK from $\mathcal{A}$ to $\mathcal{F}_{\text{EN}}$:
    send LEAK to $\mathcal{F}_{\text{EN}^+}$ and receive r
    return $\mathcal{L}(\mathsf{r})$
On message $(\text{FAKEREALITY}, \phi)$ from $\mathcal{A}$ to $\mathcal{F}_{\text{EN}}$:
    assert $\phi \in \Phi$
    send $(\text{FAKEREALITY}, \phi)$ to $\mathcal{F}_{\text{EN}^+}$
On message $(\text{CORRUPT}, U)$ from $\mathcal{A}$ to $\mathcal{F}_{\text{EN}}$:
    $\widetilde{\mathbf{U}} \leftarrow \widetilde{\mathbf{U}} \parallel U$
    send $(\text{CORRUPT}, U)$ to $\mathcal{F}_{\text{EN}^+}$ and receive $\overline{A}_U$
    for $(\alpha, \ddot{\mathsf{A}}) \in \overline{A}_U$ do
        simulate sending $(\text{KEYSHAREGEN}, \mathsf{AggS}_\alpha, \ddot{\mathsf{A}})$ to DD-FESR on behalf of $U$
On message * from $\mathcal{A}$ to $\mathcal{F}_{\text{EN}}$:
    send * to $\mathcal{F}_{\text{EN}^+}$
On message RETRIEVE from $\mathcal{A}$ to $\mathcal{F}_{\text{TBB}}$:
    return $\mathcal{T}$
```

We argue that for all messages sent by the environment, the ideal world simulator produces messages that are indistinguishable from the DoubleSteel protocol. In particular:

- SHAREEXPOSURE: when a user U shares their exposure status, $\mathcal{S}_{\text{GV}}$ is activated. If U is corrupted, it additionally receives the noisy record of U's sensitive data, u^e_{SEC}. For honest users, $\mathcal{S}_{\text{GV}}$ obtains the same sensitive records by using the LEAK function. As in the real world, DD-FESR is invoked to encrypt the sensitive data, and the resulting handle is stored in an emulated trusted bulletin board. In both worlds, the array of stored handles follows a similar distribution as they are both generated by DD-FESR for the same messages. All other behaviour of SHAREEXPOSURE is handled by $\mathcal{F}_{\text{EN}^+}$ in the same way that $\mathcal{F}_{\text{EN}}$ would.

- REGISTERANALYST: when analyst $\ddot{\mathsf{A}}$ requests permission to compute a function $\alpha \in AF$, the simulator registers them as a decryptor in DD-FESR (the same analyst can request to be registered multiple times, but the DD-FESR functionality will ignore all but the first request). $\mathcal{S}_{\text{GV}}$ then emulates a request for REGISTER-ANALYSTREQUEST for all users. For the semi-honest case, when both honest and corrupted users are allowed by the environment to accept the request for a function, they will ask DD-FESR for a KEYSHAREGEN. $\mathcal{S}_{\mathsf{GV}}$ learns about these REGISTERANALYSTACCEPT calls when either the analyst or user is corrupted. In the former case, $\mathcal{S}_{\mathsf{GV}}$ proactively sends the request to DD-FESR on behalf of the user, while in the latter it waits for the adversary to trigger the request. If both user and analyst were honest, the adversary (in either world) should not learn that a request was granted. However, given we are in the adaptive user corruption setting, the simulator has to handle keyshare generation for newly corrupted users, by requesting keyshare generation to DD-FESR for all functions they had authorized pre-corruption.

- ANALYSE: for a corrupted analyst, $\mathcal{S}_{\mathsf{GV}}$ will ensure that they sync the state of the ideal function α with that of the aggregated AggS_α in DD-FESR. To aggregate all inputs stored in its emulated trusted bulletin board $\mathcal{T}$, the analyst first encrypts the integer equal to the size of $\mathcal{T}$ and passes it for decryption to AggS_α to initialize it. Then, for all handles stored in $\mathcal{T}$, it also decrypts the corresponding values to the same aggregator. When all the decryptions have occurred, the final returned value will be the evaluation of α on the sensitive data of all users; $\mathcal{S}_{\mathsf{GV}}$ ignores this value and yields back to $\mathcal{F}_{\mathrm{EN}^+}$, which will return the ideal world result to the analyst.

  Since the inputs to the aggregator (the set of uploaded sensitive data to the trusted bulletin board) in the real world fully correspond to the input of α in the ideal world, the distributions of states and outputs for AggS_α in DD-FESR and for α in $\mathcal{F}_{\mathrm{EN}^+}$ are indistinguishable.

- LEAK: on an adversarial request to learn some values from the combination of noisy record of reality, exposed users, and corrupted users, $\mathcal{S}_{\mathsf{GV}}$ obtains the corresponding leakage r from $\mathcal{F}_{\mathrm{EN}^+}$, and filters it by the admissible leakage $\mathcal{L}$ for $\mathcal{F}_{\mathrm{EN}}$. Filtering is achieved since the condition $\mathcal{L}(\mathsf{r}) = \mathcal{L}(\mathcal{L}^+(\mathsf{r}))$ holds.

- FAKEREALITY: $\mathcal{S}_{\mathsf{GV}}$ ensures that the request to modify the noisy record of reality through $\mathcal{F}_{\mathrm{EN}^+}$ is also admissible in the real world with $\mathcal{F}_{\mathrm{EN}}$. Since the condition $\Phi \subset \Phi^+$ holds, if the ϕ is an allowable faking function in Φ, then it is also allowable in Φ^+, so the request is admissible in both the real and the ideal world.

All other messages are handled by redirecting them from $\mathcal{F}_{\mathrm{EN}^+}$ to $\mathcal{F}_{\mathrm{EN}}$, since both functionalities behave in the same manner outside the cases we have already outlined.

While the above simulator guarantees security in the semi-honest setting, it is possible to design a simulator that allows corrupted parties to diverge from the protocol. In particular, this simulator would need to handle the case of a malicious user who encrypts via DD-FESR dishonestly generated data (in that it does not match with the corrupted user's physical reality measurements). Following the lead of Canetti et al. [84], we can account for these malicious ciphertexts by using functions in Φ^+ to modify the noisy record of physical reality in $\mathcal{F}_{\mathrm{EN}^+}$. Note that this simulation strategy imposes additional deviations from the physical reality beyond those unavoidably inherited by Φ due to its own simulation needs. This makes it harder to justify the usage of the protocol by an analyst who is interested in the correctness of the data processing. $\square$

Additional remarks At a high level, the Glass-Vault protocol can support any computation in a privacy-preserving manner, in the sense that nothing beyond the computation result is revealed to the analyst; more formally, in the simulation-based model, a corrupted party's view of the protocol execution can be simulated given only its input and output. The GlassVault protocol can be considered as an *interpreter* that takes a description of any multi-input functionality along with a set of inputs, executes the functionality on the inputs and returns only the result to the analyst.

Recall that the primary reason the GlassVault protocol offers *accountability* is that it lets users have a chance to decide which computation should be executed on their sensitive data. This is of particular importance because the result of any secure computation (including functional encryption) would reveal some information about the computation's inputs. However, having such an interesting feature introduces a trade-off: if many users value their privacy and decide not to share access to their data, the analyst may not get enough to produce any useful results, foregoing the collective benefits this kind of data sharing can engender [255]. One way to solve such a dilemma would be to integrate a mechanism to incentivise users to grant access to their data for such computations (e.g., by using a blockchain token as a reward); however, even then careful considerations are required, as the framing of why a user is asked to disclose their information can impact how much they value privacy [7].

5.3.4 Performance

In the GlassVault protocol, an infected user's computation and communication complexity for the proposed data analytics purposes is independent of the total number of users, while it is linear with the number of functions requested by the analysts. An analyst's computation overhead depends on each function's complexity and the number of decryptions (as each decryption updates the function's state). The cost to non-infected users is comparable to the most efficient EN protocols that realise $\mathcal{F}_{\text{EN}}$ (such as the protocol in [84, Section 8.5]): besides passively collecting measurements of sensitive data, no other data-analytics operation is required until the ShareExposure phase.

While the costliest component of the protocol is the functional encryption module, it is possible to build an efficient implementation of GlassVault due to the construction of DoubleSteel, which relies on efficient operations facilitated by trusted hardware.

5.4 Enhancing Decentralised Contact Tracing

A contact tracing system is called centralised when a central authority is in control of a permanent identifier for every user, and enables users to exchange ephemeral identifiers with each other that only the authority can map back to the permanent identifier. A user testing positive uploads the ephemeral identifiers of people they have come into contact with to the authority, who can map the ephemeral identity of those other users to their permanent identities and notify them of infection. A second order feature of this kind of protocols is that the authority can use this information to build a general contact graph of all encounters had by infectious users.

Centralised contact tracing (CCT) systems have thus been shown to be more effective at preventing spread of diseases than their decentralised equivalent, as the contact graph can be used for various functions (among them identifying a super-spreader user with mild symptoms [154]). Much of the debate around the use of centralised servers however identified the privacy risks of putting such a contact graph at the availability of the government. Some have argued that this has to some extent been harmful for containing the pandemic [297], as the push for a privacy preserving system made the additional analysis advantages of centralised systems a secondary concern.

We now argue that Analysis-augmented Exposure Notification, as implemented by GlassVault allows to provide the same advantages of centralised contact tracing schemes, while providing greater user privacy and preserving accountability.

At a high level, centralised contact tracing computes a function that, given as input the set of contact tracing users and their location in the real world over time, returns a multi-graph where each users represents a node, and each edge is an occurrence of the two users being in proximity of each other.

Implementing Contact Graph G in $\mathsf{GlassVault}$. We now show how to support one of the main capabilities of centralised contact tracing, i.e., building contact graphs, within the GlassVault platform.

We begin by discussing about how this is achieved within the GlassVault and EN^+ formalisms, and then take a more concrete approach based on the [84] scheme as a concrete example.

Our modular approach in the previous section uses UC composition to describe GlassVault as a protocol that embeds a generic exposure notification scheme as a subroutine, without the need to exposing any of its internal. However, for some applications such as the one we will be showing next, it is necessary for the analyst to extract some information directly related to the exposure notification algorithms' implementations. Roughly, we can call α the subset of records for the physical reality functionality $\mathbb{R}$ related to what the user sends to other parties as part of the exposure notification scheme. Conversely β are the records each user receives while running the scheme. The two types of records are related together by some function γ such that $\gamma(\alpha) = \beta$ if the user that produced α was in proximity to the user that received β Exposed GlassVault users can request this kind of information to encrypt when sharing their data, so that they can be passed as the arguments of analysts' functions. We now give a concrete instantiation of what each of this parameters are in a sample contact tracing protocol, namely the $\widehat{\pi}$ protocol of Canetti et al. [84].

Let $\mathsf{chirp} \leftarrow \mathsf{Chirp}(s, \mathsf{meas})$ be a function that produces an ephemeral key from a seed; in this case, we take the set of fields A^* used to produce meas to be equal to t, the time when the chirp is produced. Chirp is an invertible function, such that there is an equivalent Chirp^{-1} such that $\mathsf{Chirp}^{-1}(s, \mathsf{Chirp}(s, \mathsf{meas})) = \mathsf{meas}$. Let K be the map of seeds each infected user used to generate their pseudonyms; and let X be a vector of triples (U, chirp, t), where U is a user, and chirp is the identifier U received at time t. Moreover, query is a set of coordinates in the adjacency matrix, i.e., two users.

```
function $\mathsf{CG}(\mathsf{X}, \mathsf{K}, \mathsf{Chirp}, \mathsf{query}, \mathsf{state})$
    if $\mathsf{G} \notin \mathsf{state}$ then
        $\mathsf{G} \leftarrow []$
        for $(U, \mathsf{chirp}, t) \in \mathsf{X}$ do
            $t_0 \leftarrow \lfloor t \rfloor$
            for $(U', s) \in \mathsf{K}$ do
                if $\mathsf{Chirp}^{-1}(s, \mathsf{chirp}) = t_0$ then
            $\mathsf{G}[U,U'] \leftarrow \mathsf{G}[U,U'] \parallel (\mathsf{chirp}, t)$
    $\mathsf{state} \leftarrow \mathsf{G}$
**else return** $\mathsf{G}[\mathsf{query}]$
```

The above function implements the centralised contact tracing contact graph.

As with any other GlassVault function, the users is able to choose whether to share their data with the protocol, and the set of all users can vote on which institutions are allowed to play the analyst role by authorizing them to execute analytics over their data. While the implemented contact graph will not be as faithful as the one implemented in a centralised contact tracing system, since some users might withhold their contact information, we argue that our design offers increased protection from the virus for privacy-conscious users. Users in countries where the only available contact tracing option is centralised might not choose to join such a system to preserve their privacy, at the cost of not being notified when they are exposed to the virus through contact with someone who is using the system. GlassVault users will instead benefit from the exposure notification system, even if they choose not to share any additional information such as the list of their contacts.

5.5 Example: infections heatmap

In this section, we provide a concrete example of a computation that a GlassVault analyst can perform: generating a daily heatmap of the current clusters of infections. This is an interesting application of GlassVault, as it relies on collecting highly sensitive location information from infected individuals.

$\mathsf{Heatmap}_{k,q}(\mathrm{x},\mathsf{s})$ is defined as a multi-input stateful function, parameterised by k: the number of distinct cells we divide the map into, and q: the minimum number of exposed users that have shared their data. The values of these parameters affect the granularity of the results, computational costs, and privacy of the exposed users. Thus, they need to be approved as part of the KEYSHAREGEN procedure (in that the parameters' values are hard-coded in the Heatmap program, so that different parameter values require different functional keys).

Given the full set of exposed users' sensitive data, the Heatmap function filters it to the exposed users' location history for the last $\mathcal{T}$ days (the maximum number of days since they might have been spreading the virus due to its incubation period), and constructs a list of $\mathcal{T} \times k$ matrices, where an entry in each matrix u contains the number of hours within a day an infected individual spent in a particular location. Location data is collected once every hour by the user's phone, and divided into k bins. The u matrix rows are in reverse chronological order, with the last row of the matrix corresponding to the locations during the most recent day and each row above in decreasing order until the first row which contains the locations $\mathcal{T}$ days ago.

Heatmap maintains as part of its state a list, m, of $\mathcal{T}$-sized circular buffers (a FIFO data structure of size $\mathcal{T}$; once more than $\mathcal{T}$ entries have been filled, the buffer starts overwriting data starting from the oldest entry). On every call with input x, the function allocates a new circular buffer b for each matrix u it constructed from x, and assigns each of u's rows to one of b's $\mathcal{T}$ elements, starting from the top row. Each element in b now contains a list of k geolocations for a specific day, with the first element containing the locations $\mathcal{T}$ days ago, and so on. For any buffer already in m, we append a new zero vector, effectively erasing the record of that user's location for the earliest day. If there is a buffer that is completely zeroed out by this operation, we remove it from m.

If we have $|\mathsf{m}| \geq q$, we return the row-wise sum of vectors $\sum_{\mathsf{b} \in \mathsf{m}} \sum_{i=0}^{\mathcal{T}-1} \mathsf{b}[i]$. The result is a single k-sized vector containing the total number of hours spent by all users within the last $\mathcal{T}$ days: our heatmap.

For simplicity of exposition, we assume that the input x is already a fully formed list of $\mathcal{T} \times k$ matrices containing a single user's location data over the last $\mathcal{T}$ days. While GlassVault functionalities typically expect a subset of $\mathbb{R}$'s noisy record of reality for fields in SEC, turning those records in a list of matrices can be delegated to the aggregator run by GlassVault to turn individual user's ciphertext into the multi-input list x.

The pseudocode below uses the following notation conventions:

- Given matrix z, the notation $z[i, j]$ denotes accessing the i-th row and j-th column of z.
- $z[i, :]$ denotes the row vector corresponding to the i-th row of z; $z[:, j]$ is the column vector corresponding to the j-th column
- We denote by CircularBuffer(n) the creation of a new n-sized circular buffer. Appending an item to the buffer is accomplished through concatenation operator $||$. After n items have been appended to a buffer, it will overwrite the first record in the buffer, and so on

```
function $\mathsf{Heatmap}_{k,q}(\mathsf{x}, \mathsf{state})$
    if $\mathsf{state} = \emptyset$ then $\mathsf{m} \leftarrow []$
    for $c \in \mathsf{m}$ do
        $c \leftarrow c \parallel \vec{0}$
        if $\forall i \in c : i = \vec{0}$ then $\mathsf{m} \leftarrow \mathsf{m} \setminus c$
    for $u \in \mathsf{x}$ do
        $b \leftarrow \mathsf{CircularBuffer}(\mathcal{T})$
        for $\{i = 0; i < \mathcal{T}; i{+}{+}\}$ do
            assert $\sum_{j=0}^{k-1} u[i, j] = 24$
            $b \leftarrow b \parallel u[i, :]$
        $\mathsf{m} \leftarrow \mathsf{m} \parallel b$
    $\mathsf{y} \leftarrow \vec{0}$
    if $|\mathsf{m}| \geq q$ then
        for $u \in \mathsf{m}$ do
            for $\{i = 0; i < \mathcal{T}; i{+}{+}\}$ do
                $\mathsf{y} \leftarrow \mathsf{y} + u[i, :]$
    return $\mathsf{y}$
```

For the results' correctness, an analyst should run the function (through a decryption operation) once a day. As the summation of user location vectors is a destructive operation, the probability that a malicious analyst can recover any specific user's input will be inversely proportional to q.

The security proof of Theorem 5.2 is either in the semi-honest setting or requires a fake reality function in which a user can tamper with their own client applications to upload malicious data (terrorist attack [287]). Since there is no secure pipeline from the raw measurements from sensors to a specific application, unless we adopt the strong requirement that every client also runs a TEE (as in [149]) or uses their device hardware Root-of-trust to certify the authenticity of peripheral readings (as in [243]), it is impossible to certify that the users' inputs are valid. Like most other remote computation systems, GlassVault cannot provide blanket protection against this kind of attack. However, due to its generality, GlassVault allows analysts to use functions that include "sanity checks" to ensure that the data being uploaded are at least sensible, in order to limit the damage that the attack may cause. In the heatmap case, one such check could be verifying that for each row of u, it must hold that its column-wise sum is equal to 24, since each row represents the number of hours spent across various locations by the user in a day (assuming the user's phone is on and able to collect their location at least once an hour throughout a day). To capture this type of attack in the ideal functionality $\mathcal{F}_{\mathrm{EN}^+}$, we instantiate it with a FAKEREALITY function in Φ^+ such that, if a malicious user U uploads this type of fake geolocation, it will update U's position within the noisy record of physical reality to match U's claimed location, while making sure that other users who compute risk exposure and have been in close contact with U will still be notified.

We highlight that Bruni et al. [66] propose an ad-hoc scheme that produces similar output. Their scheme relies on combining infection data provided by health authorities with the mass collection of cellphone location data from mobile phone operators. Unlike GlassVault, the approach in [66] does not support any mechanism that allows the subjects of data collection to provide their direct consent and opt-out of the computation.

Chapter 6

Conclusion

At this point SGX is just so broken that it seems like its only purpose is to provide PhD students something to write a paper on :)

Usmann Khan

This thesis reflects research on formalising Trusted Execution Environments spanning the last five years, a time marked by significant changes in Confidential Computing. The technology has solidified its position as a cloud computing solution, shifting away from consumer markets, with SGX being discontinued for any consumer-grade Intel CPU. Hardware-assisted security might have increased its role in the consumer market, but only in the form of the less powerful TPMs, which are now required to run the latest version of the Windows operating system [209]. While we welcome the increased usage of TEEs in the cloud setting as a promising development to increase accountability in cloud computing vendors and applications, another other major trend since the start of our work should cause some concern. Due to the difficulty of developing secure enclave programs, the industry has shifted from process-based isolation to virtualisation, allowing unmodified application to run on the current generation of Confidential Computing hardware. With the complexity of TEE implementations, it is difficult enough to reason about the small Trusted Computing Base of enclaves for realising secure applications as we have done in this work. Trusting the entirety of a virtualised operating system opens the door to a much larger number of exploits than those we already have.

Over the last few chapters, we have gone through various stages of the lifecycle of a protocol that relies on TEEs, from defining Steel and proving its security under UC in Chapter 3, to showing how to use realise a privacy-preserving in Chapter 5, via a significant detour to revisite the foundations on how we model TEEs cryptographically in Chapter 4. We now make some observation about limitations of our work, and future research directions for each chapter.

Chapter 2: Background As a first observation, we note that our background chapter includes pointers to several surveys on TEEs, from their architecture to application and attacks. However, we find two gaps in the existing literature: a lack of surveys on protection mechanisms against TEE attacks, and one on formalising TEEs. We have attempted to address the latter in our Section 2.2.4, but further work is needed to establish what the relationship between the listed work are, and what elements of what tools are available to approach formalisation at each level of the TEE development lifecycle (including the gaps). Our Section 2.2.3.1 provides an overview of protection mechanisms for a specific class of attacks, but we believe that a more comprehensive survey for additional vulnerabilities and defences is necessary.

Chapter 3: Steel Our work on Steel was accepted as part of the 2021 IACR Public Key Cryptography Conference, and published in its proceeding as [53]. Despite undergoing the process of review, in the very last stages of compiling this thesis, we became aware of a distinguishing attack between the real and ideal world.

The attack stems from our formulation of functions in the ideal world diverging from how enclave programs manage functional key provisioning in the real world. The ideal functionality FESR stores the state of each function for which it had produced a decryption key in a table, indexed by the party that received the key and the function. If a decryptor requests the generation of a functional key for a function it already has access to, it will overwrite the value of the function state on the table with a new initial state. In the real world, a decryptor party could choose to install multiple copies of a functional enclave, and request authorisation to compute that function from the decryption enclave. If the party has successfully been granted a functional key, the decryption enclave will authorise all copies of the functional enclave for that party to decrypt under that function. This discrepancy does not lead to a forking attack, as the attestation signatures encode the enclave ID, so the adversary could not claim that the outputs of two enclaves were produced by the same function, nor can they fork the enclave at any arbitrary point in execution (unless it is a deterministic function, and the adversary replays all of its input to the second copy until the desired state). The attack is primarily a modelling issue, and does not affect regular Functional encryption or Iron due to their lack of state.

It is not obvious what a desirable fix should be; as it stands, both the ideal functionality and the protocol seem flawed. One option would be to modify the ideal functionality as to allow a decryptor to maintain multiple copies of the same function, to capture the behaviour of the protocol. Alternatively, both ideal functionality and the protocol could only allow one copy of each function: this would require modifying the functionality, by checking that $\mathcal{P}[\mathsf{B},\mathsf{F}]$ is not empty to avoid overwriting it with $\emptyset$, and the decryption enclave in Steel by similarly keeping track of previous key generation requests. Finally, we could choose to preserve the current ideal functionality behaviour by allowing a decryptor to reset the function state when requesting a new key. This would require modifying Steel to either force the functional enclave to erase its memory on a new keygen request, or to go through key provisioning after every run execution, as to check with $\mathsf{prog}_{\mathsf{DE}}$ that it still is the only authorised enclave to compute the function. Having a liveness check for each decryption enclave might be a welcome change in the Steel architecture, as it would allow us to augment the protocol with rollback protection using the technique described in the next paragraph. While it is possible to apply one of these fixes and adjust the simulator accordingly (since the adversary is notified on every KEYGEN message), some considerations are required on how this would affect the DD-FESR variant and its simulator, which uses the FESR simulator in a black box way.

Chapter 4: Modelling Our new models for global attestation captures existing functionalities in the literature, as well as other variants of TEEs not previously proven in UC. While we showed somewhat informally how to capture these using our language of oracles, our work requires further validation in proving that our setups can replace the existing ones, and that the formalisation is robust enough to be adopted by other researchers for their own TEE-hybrid protocols. At the risk of being overly ambitious, we believe that G_{att}^{mod} should become the canonical functionality used in UC proofs. This would serve the dual purpose of forcing the protocol designer to think about the feature requirements of the enclave and what attacks would be allowable in their settings, while also providing theorems to realise their version of the setup from weaker versions that we are more likely to develop. The final goal of this modelling project would be to formalise a real TEE implementation, and to produce a chain of wrapper protocols that can lead us from this version of G_{att}^{mod} to G_{att}^{PST}, or even stronger versions. It seems like much more bridging work will be required between the cryptography and system communities to produce such a model, and while this work has not yet found a venue for publication, we have seen some early interest in this line of work from members of the formal method community who specialise in formalising TEEs, with an early version of this work being accepted to the 3rd PAVeTrust workshop.

As a first step, further examples of how to apply theorems 4.1 and 4.2 will be needed. In Section 4.5, we presented an example of a wrapper protocol to remove the Rollback attack interface from G_{att}^{mod} by using the Store, Fetch oracle interfaces presented in Section 4.3.3. While this example helped us address the attack on Steel introduced earlier in the chapter, it is not entirely satisfying for two reasons. First, it is an extremely simple protocol that does not involve any assisting enclaves, and thus does not serve as a good example of a security proof with a more complex simulator that needs to handle the interaction of multiple enclaves running on a combination of trusted and untrusted parties. Furthermore, it does not really provide a satisfying solution to rollback attacks in the real world. Preventing rollback attacks in a setting where we have access to *some* trusted storage is a straighforward idea, but no realistic TEE has managed to achieve this property. In Section 2.2.3, we examined how existing trusted protection mechanisms developed on trusted counters are generally easy to bypass, and that multiple solutions have been designed to provide a distributed rollback protection mechanism. Since most of these works do not originate from the cryptographic community, they often lack a clear treatment of the network and adversarial model, or a satisfying proof of security. A likely target for future work is providing a new proof of equivalence to remove Rollback attacks from $G_{\mathtt{att}}^{mod}$ through one of those protocols, without relying on local trusted storage. It should be possible to easily port protocol $\mathcal{W}$ from Section 4.5 onto a shell with access to a distributed storage functionality, such as the one presented in Section 4.3.6. We believe it should then be possible to show that one of the rollback protection protocols in the literature (or a new protocol that combines some of their features) can UC-realise the registry functionality (or similar).

There are further opportunities for future work on the $G_{\mathtt{att}}^{mod}$ model, in particular building on the formalisation of our TEE Zoo of Section 4.3 with further features and attacks. In particular, given recent interest in GPU accelleration due to the commercial implications of machine learning, formalising enclaves with access to an oracle for a heterogenous computing device might enable the development of much needed privacy and safety protocols in the field. Additionally, there is room to explore a larger class of corruptions from the TEE literature, including partial corruption (where only some enclaves running on a party can be accessed by the adversary, a typical setting in the cloud computing scenario) or attacks that allow interrupting the enclave at a more granular level of execution than a subroutine. Further work is also needed in analysing different types of attestation protocols, and how they affect our modelling and equivalence theorems.

Finally, we might want to consider additional models of TEEs that allow enclaves with different feature (or attacker) oracle sets to co-exist. The motivation for providing this extension would be performance, as some of the oracles required to implement a feature (or defend against an attack) might cause an undesirable performance penalty. It might actually be desirable for the purposes of real world realisations to show that a smaller set of more powerful (and expensive) enclaves can provide security for a larger number of enclaves through a protocol. To go back to our running example, it is possible to modify Steel to become rollback resilient by simply running a rollback resilient Decryption Enclave, even if the Functional Enclaves are susceptible to rollback attacks. Such a protocol would leverage the same ideas as the Section 4.5 protocol, but use the Decryption Enclave as its trusted storage implementation.

Chapter 5: GlassVault There are several possible directions for future research in relation to GlassVault. An immediate goal would be to provide an implementation, as well as some sample data analytics workloads; examine their run-time, and optimise the system's bottlenecks. Since GlassVault's analytics results could influence public policy, it is interesting to investigate how this platform could be equipped with mechanisms that allow to non-repudiably verify the authenticity of the output of analysts. Currently functional decryption results are deniable by design. Another appealing future research direction is to provide a more fine-grained approach to our centralised contact tracing implementation in GlassVault. The functional encryption scheme could enforce, for example, a threshold version of the contact graph application, enforcing a user defined "privacy budget" on how many query the analyst may run. More broadly, DoubleSteel could be used for other data science tasks that require large scale collection of user data, something that we are likely to see more of in the future, especially in a time of crisis. It is crucial that we refine the design and test deployment of such tools at a time when discussions about the values of privacy and consent and its tradeoffs with public utility can be discussed without the urgency of a global emergency.

Bibliography

[1] https://security.apple.com/blog/private-cloud-compute/. June 2024.

[2] Martín Abadi and Phillip Rogaway. “Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)”. In: *Journal of Cryptology* 15.2 (Mar. 2002), pp. 103–127. DOI: 10.1007/s00145-001-0014-7.

[3] Roba Abbas and Katina Michael. “COVID-19 contact trace app deployments: learnings from Australia and Singapore”. In: *IEEE Consumer Electronics Magazine* 9.5 (2020), pp. 65–70.

[4] Michel Abdalla et al. “Decentralizing Inner-Product Functional Encryption”. In: *PKC 2019: 22nd International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. Lecture Notes in Computer Science. Beijing, China: Springer, Cham, Switzerland, Apr. 2019, pp. 128–157. DOI: 10.1007/978-3-030-17259-6_5.

[5] Bhavani M. Thuraisingham et al., eds. *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Dallas, TX, USA: ACM Press, Oct. 2017.

[6] Heng Yin et al., eds. *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Los Angeles, CA, USA: ACM Press, Nov. 2022.

[7] Alessandro Acquisti, Leslie K John, and George Loewenstein. “What is privacy worth?” In: *The Journal of Legal Studies* 42.2 (2013), pp. 249–274.

[8] Shashank Agrawal and David J. Wu. “Functional Encryption: Deterministic to Randomized Functions from Simple Assumptions”. In: *Advances in Cryptology – EUROCRYPT 2017, Part II*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10211. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 30–61. DOI: 10.1007/978-3-319-56614-6_2.

[9] Shweta Agrawal et al. *Functional Encryption: New Perspectives and Lower Bounds*. Cryptology ePrint Archive, Report 2012/468. https://eprint.iacr.org/2012/468. 2012.

[10] Adil Ahmad et al. "OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX". In: *ISOC Network and Distributed System Security Symposium – NDSS 2019*. San Diego, CA, USA: The Internet Society, Feb. 2019. DOI: 10.14722/ndss.2019.23513.

[11] Danel Ahman et al. "Recalling a witness: foundations and applications of monotonic state". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3158153. URL: http://dx.doi.org/10.1145/3158153.

[12] Nadeem Ahmed et al. "A Survey of COVID-19 Contact Tracing Apps". In: *IEEE Access* 8 (2020), pp. 134577–134601. DOI: 10.1109/ACCESS.2020.3010226. URL: https://doi.org/10.1109/ACCESS.2020.3010226.

[13] Fraunhofer AISEC. *Pandemic Contact Tracing Apps: DP-3T, PEPP-PT NTK, and ROBERT from a Privacy Perspective*. Cryptology ePrint Archive, Report 2020/489. https://eprint.iacr.org/2020/489. 2020.

[14] UK Health Data Research Alliance and NHSX. *Building Trusted Research Environments - Principles and Best Practices; Towards TRE ecosystems*. 2021.

[15] Hannah Alsdurf et al. "Covi White Paper". In: *CoRR* (2020).

[16] Tehilla Shwartz Altshuler and Rachel Aridor Hershkowitz. *How Isreal's COVID-19 mass surveillance operation works*. 2020.

[17] Joël Alwen, abhi shelat, and Ivan Visconti. "Collusion-Free Protocols in the Mediated Model". In: *Advances in Cryptology – CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2008, pp. 497–514. DOI: 10.1007/978-3-540-85174-5_28.

[18] Ross J. Anderson. *'Trusted Computing' Frequently Asked Questions*. 2003. URL: https://www.cl.cam.ac.uk/~rja14/tcpa-faq.html.

[19] Ross J. Anderson. “Cryptography and competition policy: issues with ’trusted computing’”. In: *22nd ACM Symposium Annual on Principles of Distributed Computing*. Ed. by Elizabeth Borowsky and Sergio Rajsbaum. Boston, MA, USA: Association for Computing Machinery, July 2003, pp. 3–10. DOI: 10.1145/872035.872036.

[20] Ross J. Anderson. “Why cryptosystems fail”. In: *Communications of the ACM* 37.11 (Nov. 1994), pp. 32–40. ISSN: 1557-7317. DOI: 10.1145/188280.188291. URL: http://dx.doi.org/10.1145/188280.188291.

[21] Sebastian Angel et al. “Nimble: Rollback Protection for Confidential Cloud Services”. In: *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*. Ed. by Roxana Geambasu and Ed Nightingale. USENIX Association, 2023, pp. 193–208. URL: https://www.usenix.org/conference/osdi23/presentation/angel.

[22] Pedro Antonino, Ante Derek, and Wojciech Aleksander Wołoszyn. *Flexible remote attestation of pre-SNP SEV VMs using SGX enclaves*. 2023. arXiv: 2305.09351v1 [cs.CR].

[23] Pedro Antonino, Wojciech Aleksander Woloszyn, and A. W. Roscoe. “Guardian: Symbolic Validation of Orderliness in SGX Enclaves”. In: *Proceedings of the 2021 on Cloud Computing Security Workshop*. CCS ’21. ACM, Nov. 2021. DOI: 10.1145/3474123.3486755. URL: http://dx.doi.org/10.1145/3474123.3486755.

[24] Apple and Google. *Exposure Notification API*. https://www.google.com/covid19/exposurenotifications/. 2020.

[25] William A. Arbaugh, David J. Farber, and Jonathan M. Smith. “A Secure and Reliable Bootstrap Architecture”. In: *1997 IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Computer Society Press, 1997, pp. 65–71. DOI: 10.1109/SECPRI.1997.601317.

[26] Ghada Arfaoui, Said Gharout, and Jacques Traore. “Trusted Execution Environments: A Look under the Hood”. In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, Apr. 2014. DOI: 10.1109/mobilecloud.2014.47. URL: http://dx.doi.org/10.1109/MobileCloud.2014.47.

[27] Sergei Arnautov et al. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 689–703. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`.

[28] T. W. Arnold and L. P. Van Doorn. “The IBM PCIXCC: a New Cryptographic Coprocessor for the IBM eServer”. In: *IBM Journal of Research and Development* 48.3.4 (2004), pp. 475–487. DOI: `10.1147/rd.483.0475`. URL: `http://dx.doi.org/10.1147/rd.483.0475`.

[29] JP Aumasson and Luis Merino. “SGX secure enclaves in practice: security and crypto review”. In: *Black Hat* 2016 (2016), p. 10.

[30] Gennaro Avitabile et al. *Towards Defeating Mass Surveillance and SARS-CoV-2: The Pronto-C2 Fully Decentralized Automatic Contact Tracing System*. Cryptology ePrint Archive, Report 2020/493. `https://eprint.iacr.org/2020/493`. 2020.

[31] Michael Backes, Birgit Pfitzmann, and Michael Waidner. “A General Composition Theorem for Secure Reactive Systems”. In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2004, pp. 336–354. DOI: `10.1007/978-3-540-24638-1_19`.

[32] Michael Backes, Birgit Pfitzmann, and Michael Waidner. *The Reactive Simulatability (RSIM) Framework for Asynchronous Systems*. Cryptology ePrint Archive, Report 2004/082. `https://eprint.iacr.org/2004/082`. 2004.

[33] Christian Badertscher, Julia Hesse, and Vassilis Zikas. *On the (Ir)Replaceability of Global Setups, or How (Not) to Use a Global Ledger*. Cryptology ePrint Archive, Report 2020/1489. `https://eprint.iacr.org/2020/1489`. 2020.

[34] Christian Badertscher, Julia Hesse, and Vassilis Zikas. “On the (Ir)Replaceability of Global Setups, or How (Not) to Use a Global Ledger”. In: *TCC 2021: 19th Theory of Cryptography Conference, Part II*. Ed. by Kobbi Nissim and Brent Waters. Vol. 13043. Lecture Notes in Computer Science. Raleigh, NC, USA: Springer, Cham, Switzerland, Nov. 2021, pp. 626–657. DOI: `10.1007/978-3-030-90453-1_22`.

[35] Christian Badertscher et al. *Consistency for Functional Encryption*. Cryptology ePrint Archive, Report 2020/137. https://eprint.iacr.org/2020/137. 2020.

[36] Christian Badertscher et al. "Universal Composition with Global Subroutines: Capturing Global Setup Within Plain UC". In: *TCC 2020: 18th Theory of Cryptography Conference, Part III*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12552. Lecture Notes in Computer Science. Durham, NC, USA: Springer, Cham, Switzerland, Nov. 2020, pp. 1–30. DOI: 10.1007/978-3-030-64381-2_1.

[37] Christian Badertscher et al. *Universal Composition with Global Subroutines: Capturing Global Setup within plain UC*. Cryptology ePrint Archive, Report 2020/1209. https://eprint.iacr.org/2020/1209. 2020.

[38] Karim Baghery et al. *Another Look at Extraction and Randomization of Groth's zk-SNARK*. Cryptology ePrint Archive, Report 2020/811. https://eprint.iacr.org/2020/811. 2020.

[39] Raad Bahmani et al. "CURE: A Security Architecture with CUstomizable and Resilient Enclaves". In: *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Ed. by Michael D. Bailey and Rachel Greenstadt. USENIX Association, 2021, pp. 1073–1090. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/bahmani.

[40] Raad Bahmani et al. *Secure Multiparty Computation from SGX*. Cryptology ePrint Archive, Report 2016/1057. https://eprint.iacr.org/2016/1057. 2016.

[41] Annette Baier. "Trust and Antitrust". In: *Ethics* 96.2 (Jan. 1986), pp. 231–260. ISSN: 1539-297X. DOI: 10.1086/292745. URL: http://dx.doi.org/10.1086/292745.

[42] Maurice Bailleu et al. "Avocado: A Secure In-Memory Distributed Storage System". In: *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. Ed. by Irina Calciu and Geoff Kuenning. USENIX Association, 2021, pp. 65–79. URL: https://www.usenix.org/conference/atc21/presentation/bailleu.

[43] Maurice Bailleu et al. “SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution”. In: *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. Ed. by Arif Merchant and Hakim Weatherspoon. USENIX Association, 2019, pp. 173–190. URL: https://www.usenix.org/conference/fast19/presentation/bailleu.

[44] Alexandros Bakas and Antonis Michalas. *It Runs and it Hides: A Function-Hiding Construction for Private-Key Multi-Input Functional Encryption*. Cryptology ePrint Archive, Report 2023/024. https://eprint.iacr.org/2023/024. 2023.

[45] Ero Balsa, Helen Nissenbaum, and Sunoo Park. “Cryptography, Trust and Privacy: It’s Complicated”. In: *Proceedings of the 2022 Symposium on Computer Science and Law*. CSLAW ’22. ACM, Nov. 2022. DOI: 10.1145/3511265.3550443. URL: http://dx.doi.org/10.1145/3511265.3550443.

[46] Boaz Barak et al. “Universally Composable Protocols with Relaxed Set-Up Assumptions”. In: *45th Annual Symposium on Foundations of Computer Science*. Rome, Italy: IEEE Computer Society Press, Oct. 2004, pp. 186–195. DOI: 10.1109/FOCS.2004.71.

[47] Manuel Barbosa et al. *Foundations of Hardware-Based Attested Computation and Application to SGX*. Cryptology ePrint Archive, Report 2016/014. https://eprint.iacr.org/2016/014. 2016.

[48] Manuel Barbosa et al. “SoK: Computer-Aided Cryptography”. In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008.

[49] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 267–283. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann.

[50] Saskia Bayreuther et al. “Hidden Δ-fairness: A Novel Notion for Fair Secure Two-Party Computation”. In: *IACR Cryptol. ePrint Arch.* (2024), p. 587. URL: `https://eprint.iacr.org/2024/587`.

[51] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. “Hybrids on Steroids: SGX-Based High Performance BFT”. In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic. ACM, 2017, pp. 222–237. ISBN: 978-1-4503-4938-3. DOI: `10.1145/3064176.3064213`. URL: `https://doi.org/10.1145/3064176.3064213`.

[52] Robert M Best. “Preventing software piracy with crypto-microprocessors”. In: *Proceedings of IEEE Spring COMPCON*. Vol. 80. 1980, pp. 466–469.

[53] Pramod Bhatotia et al. “Steel: Composable Hardware-Based Stateful and Randomised Functional Encryption”. In: *PKC 2021: 24th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Juan Garay. Vol. 12711. Lecture Notes in Computer Science. Virtual Event: Springer, Cham, Switzerland, May 2021, pp. 709–736. DOI: `10.1007/978-3-030-75248-4_25`.

[54] Pramod Bhatotia et al. *Steel: Composable Hardware-based Stateful and Randomised Functional Encryption*. Cryptology ePrint Archive, Report 2021/269. `https://eprint.iacr.org/2021/269`. 2021.

[55] Jean-François Biasse et al. *Trace-Σ: a privacy-preserving contact tracing app*. Cryptology ePrint Archive, Report 2020/792. `https://eprint.iacr.org/2020/792`. 2020.

[56] Henk Birkholz et al. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023. DOI: `10.17487/RFC9334`. URL: `https://www.rfc-editor.org/info/rfc9334`.

[57] Bruno Blanchet. “Security Protocol Verification: Symbolic and Computational Models”. In: *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*. Ed. by Pierpaolo Degano and Joshua D. Guttman. Vol. 7215. Lecture Notes in Computer Science. Springer, 2012, pp. 3–29. DOI: `10.1007/`

978-3-642-28641-4_2. URL: https://doi.org/10.1007/978-3-642-28641-4%5C_2.

[58] Márton Bognár, Jo Van Bulck, and Frank Piessens. “Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures”. In: *2022 IEEE Symposium on Security and Privacy (SP)* (2022), pp. 1638–1655. URL: https://api.semanticscholar.org/CorpusID:251141162.

[59] Dan Boneh, Amit Sahai, and Brent Waters. “Functional Encryption: Definitions and Challenges”. In: *TCC 2011: 8th Theory of Cryptography Conference*. Ed. by Yuval Ishai. Vol. 6597. Lecture Notes in Computer Science. Providence, RI, USA: Springer, Berlin, Heidelberg, Germany, Mar. 2011, pp. 253–273. DOI: 10.1007/978-3-642-19571-6_16.

[60] Thomas Bourgeat et al. “MI6: Secure Enclaves in a Speculative Out-of-Order Processor”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 42–56. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358310. URL: https://doi.org/10.1145/3352460.3358310.

[61] Elette Boyle, Kai-Min Chung, and Rafael Pass. “On Extractability Obfuscation”. In: *TCC 2014: 11th Theory of Cryptography Conference*. Ed. by Yehuda Lindell. Vol. 8349. Lecture Notes in Computer Science. San Diego, CA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2014, pp. 52–73. DOI: 10.1007/978-3-642-54242-8_3.

[62] Marcus Brandenburger et al. “Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory”. In: *CoRR* (2017). arXiv: 1701.00981 [cs.DC]. URL: http://arxiv.org/abs/1701.00981v2.

[63] Marcus Brandenburger et al. “Trusted Computing Meets Blockchain: Rollback Attacks and a Solution for Hyperledger Fabric”. In: *38th Symposium on Reliable Distributed Systems, SRDS 2019, Lyon, France, October 1-4, 2019*. IEEE, 2019, pp. 324–333. ISBN: 978-1-7281-4222-7. DOI: 10.1109/SRDS47363.2019.00045. URL: https://doi.org/10.1109/SRDS47363.2019.00045.

[64] Konstantinos Brazitikos and Vassilis Zikas. "General Adversary Structures in Byzantine Agreement and Multi-Party Computation with Active and Omission Corruption". In: *IACR Cryptol. ePrint Arch.* (2024), p. 209. URL: `https://eprint.iacr.org/2024/209`.

[65] Samira Briongos et al. "No Forking Way: Detecting Cloning Attacks on Intel SGX Applications". In: *Annual Computer Security Applications Conference, ACSAC 2023, Austin, TX, USA, December 4-8, 2023*. ACM, 2023, pp. 744–758. DOI: `10.1145/3627106.3627187`. URL: `https://doi.org/10.1145/3627106.3627187`.

[66] Alessandro Bruni et al. *Privately Connecting Mobility to Infectious Diseases via Applied Cryptography*. Cryptology ePrint Archive, Report 2020/522. `https://eprint.iacr.org/2020/522`. 2020.

[67] Maja Hojer Bruun, Astrid Oberborbeck Andersen, and Adrienne Mannov. "Infrastructures of trust and distrust: The politics and ethics of emerging cryptographic technologies". In: *Anthropology Today* 36.2 (2020), pp. 13–17.

[68] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017, 4:1–4:6. ISBN: 978-1-4503-5097-6. DOI: `10.1145/3152701.3152706`. URL: `https://doi.org/10.1145/3152701.3152706`.

[69] Matteo Busi, Riccardo Focardi, and Flaminia Luccio. *Bridging the Gap: Automated Analysis of Sancus*. 2024. arXiv: `2404.09518v1 [cs.CR]`.

[70] Matteo Busi et al. "Securing Interruptible Enclaved Execution on Small Microprocessors". In: *ACM Trans. Program. Lang. Syst.* 43.3 (2021), 12:1–12:77. DOI: `10.1145/3470534`. URL: `https://doi.org/10.1145/3470534`.

[71] Jan Camenisch, Manu Drijvers, and Björn Tackmann. *Multi-Protocol UC and its Use for Building Modular and Efficient Protocols*. Cryptology ePrint Archive, Report 2019/065. `https://eprint.iacr.org/2019/065`. 2019.

[72] Jan Camenisch et al. *iUC: Flexible Universal Composability Made Simple*. Cryptology ePrint Archive, Report 2019/1073. `https://eprint.iacr.org/2019/1073`. 2019.

[73] L. Jean Camp, Helen Nissenbaum, and Cathleen McGrath. "Trust: A Collision of Paradigms". In: *FC 2001: 5th International Conference on Financial Cryptography*. Ed. by Paul F. Syverson. Vol. 2339. Lecture Notes in Computer Science. Grand Cayman, British West Indies: Springer, Berlin, Heidelberg, Germany, Feb. 2002, pp. 91–105. DOI: 10.1007/3-540-46088-8_10.

[74] Ran Canetti. "Security and Composition of Multiparty Cryptographic Protocols". In: *Journal of Cryptology* 13.1 (Jan. 2000), pp. 143–202. DOI: 10.1007/s001459910006.

[75] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols*. Cryptology ePrint Archive, Report 2000/067. https://eprint.iacr.org/2000/067. 2000.

[76] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd Annual Symposium on Foundations of Computer Science*. Las Vegas, NV, USA: IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[77] Ran Canetti. *Universally Composable Signatures, Certification and Authentication*. Cryptology ePrint Archive, Report 2003/239. https://eprint.iacr.org/2003/239. 2003.

[78] Ran Canetti, Asaf Cohen, and Yehuda Lindell. "A Simpler Variant of Universally Composable Security for Standard Multiparty Computation". In: *Advances in Cryptology – CRYPTO 2015, Part II*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9216. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2015, pp. 3–22. DOI: 10.1007/978-3-662-48000-7_1.

[79] Ran Canetti and Marc Fischlin. "Universally Composable Commitments". In: *Advances in Cryptology – CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2001, pp. 19–40. DOI: 10.1007/3-540-44647-8_2.

[80] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. "On the Limitations of Universally Composable Two-Party Computation Without Set-Up Assumptions". In: *Journal of Cryptology* 19.2 (Apr. 2006), pp. 135–167. DOI: 10.1007/s00145-005-0419-9.

[81] Ran Canetti and Tal Rabin. “Universal Composition with Joint State”. In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2003, pp. 265–281. DOI: 10.1007/978-3-540-45146-4_16.

[82] Ran Canetti, Daniel Shahaf, and Margarita Vald. “Universally Composable Authentication and Key-Exchange with Global PKI”. In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part II*. Ed. by Chen-Mou Cheng et al. Vol. 9615. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Berlin, Heidelberg, Germany, Mar. 2016, pp. 265–296. DOI: 10.1007/978-3-662-49387-8_11.

[83] Ran Canetti, Ari Trachtenberg, and Mayank Varia. *Anonymous Collocation Discovery: Harnessing Privacy to Tame the Coronavirus*. 2020. arXiv: 2003.13670 [cs.CY].

[84] Ran Canetti et al. *Privacy-Preserving Automated Exposure Notification*. Cryptology ePrint Archive, Report 2020/863. https://eprint.iacr.org/2020/863. 2020.

[85] Ran Canetti et al. *Universally Composable End-to-End Secure Messaging*. Cryptology ePrint Archive, Report 2022/376. https://eprint.iacr.org/2022/376. 2022.

[86] Ran Canetti et al. “Universally Composable End-to-End Secure Messaging”. In: *Advances in Cryptology – CRYPTO 2022, Part II*. Ed. by Yevgeniy Dodis and Thomas Shrimpton. Vol. 13508. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2022, pp. 3–33. DOI: 10.1007/978-3-031-15979-4_1.

[87] Ran Canetti et al. “Universally Composable Security with Global Setup”. In: *TCC 2007: 4th Theory of Cryptography Conference*. Ed. by Salil P. Vadhan. Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer, Berlin, Heidelberg, Germany, Feb. 2007, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4.

[88] Ran Canetti et al. “Using Universal Composition to Design and Analyze Secure Complex Hardware Systems”. In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13,*

2020. IEEE, 2020, pp. 520–525. ISBN: 978-3-9819263-4-7. DOI: 10.23919/DATE48585.2020.9116295. URL: https://doi.org/10.23919/DATE48585.2020.9116295.

[89] DG Carikli and M Blanc. *The Intel Management Engine: An Attack on Computer Users' Freedom*. Tech. rep. Free Software Foundation, 2018.

[90] Shanwei Cen and Bo Zhang. *Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services*. Online at: https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf. 2017.

[91] Centers for Disease Control and Prevention. *Contact Tracing*. https://www.cdc.gov/coronavirus/2019-ncov/daily-life-coping/contact-tracing.html. 2021.

[92] Nishanth Chandran et al. *Functional Encryption: Decentralised and Delegatable*. Cryptology ePrint Archive, Report 2015/1017. https://eprint.iacr.org/2015/1017. 2015.

[93] Gauthier Chassang. "The impact of the EU general data protection regulation on scientific research". In: *ecancermedicalscience* 11 (2017).

[94] Raymond Cheng et al. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution". In: *CoRR* abs/1804.05141 (2018). arXiv: 1804.05141. URL: http://arxiv.org/abs/1804.05141.

[95] Raymond Cheng et al. "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts". In: *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 185–200. ISBN: 978-1-7281-1148-3. DOI: 10.1109/EuroSP.2019.00023. URL: https://doi.org/10.1109/EuroSP.2019.00023.

[96] Jérémy Chotard et al. "Decentralized Multi-Client Functional Encryption for Inner Product". In: *Advances in Cryptology – ASIACRYPT 2018, Part II*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11273. Lecture Notes in Computer Science. Brisbane, Queensland, Australia: Springer, Cham, Switzerland, Dec. 2018, pp. 703–732. DOI: 10.1007/978-3-030-03329-3_24.

[97] Jérémy Chotard et al. “Dynamic Decentralized Functional Encryption”. In: *Advances in Cryptology – CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2020, pp. 747–775. DOI: 10.1007/978-3-030-56784-2_25.

[98] Arka Rai Choudhuri et al. “Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards”. In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 719–728. DOI: 10.1145/3133956.3134092.

[99] Byung-Gon Chun et al. “Attested append-only memory: making adversaries stick to their word”. In: *Symposium on Operating Systems Principles*. 2007. URL: https://api.semanticscholar.org/CorpusID:6685352.

[100] Kai-Min Chung, Jonathan Katz, and Hong-Sheng Zhou. “Functional Encryption from (Small) Hardware Tokens”. In: *Advances in Cryptology – ASIACRYPT 2013, Part II*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. Lecture Notes in Computer Science. Bengalore, India: Springer, Berlin, Heidelberg, Germany, Dec. 2013, pp. 120–139. DOI: 10.1007/978-3-642-42045-0_7.

[101] Michele Ciampi, Aggelos Kiayias, and Yu Shen. “Universal Composable Transaction Serialization with Order Fairness”. In: *44th Annual International Cryptology Conference*. 2024.

[102] Michele Ciampi, Yun Lu, and Vassilis Zikas. *Collusion-Preserving Computation without a Mediator*. Cryptology ePrint Archive, Report 2020/497. https://eprint.iacr.org/2020/497. 2020.

[103] Michele Ciampi, Yun Lu, and Vassilis Zikas. “Collusion-Preserving Computation without a Mediator”. In: *CSF 2022: IEEE 35th Computer Security Foundations Symposium*. Haifa, Israel: IEEE Computer Society Press, Aug. 2022, pp. 211–226. DOI: 10.1109/CSF54842.2022.9919678.

[104] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016.

[105] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *USENIX Security*

2016: 25th USENIX Security Symposium. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 857–874.

[106] Max Crone. “Towards attack-tolerant trusted execution environments”. MA thesis. Aalto University, 2021.

[107] Joe Kilian, ed. *Advances in Cryptology – CRYPTO 2001*. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2001.

[108] George Danezis. “Introduction to privacy technology”. In: *Katholieke University Leuven, COSIC: Leuven, Belgium* (2007).

[109] Poulami Das et al. “FastKitten: Practical Smart Contracts on Bitcoin”. In: *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, 2019, pp. 801–818. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/das.

[110] Alfredo De Santis et al. “Robust Non-interactive Zero Knowledge”. In: *Advances in Cryptology – CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 2001, pp. 566–598. DOI: 10.1007/3-540-44647-8_33.

[111] Jérémie Decouchant et al. “DAMYSUS: streamlined BFT consensus leveraging trusted components”. In: *EuroSys ’22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*. Ed. by Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis. ACM, 2022, pp. 1–16. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519568. URL: https://doi.org/10.1145/3492321.3519568.

[112] Stéphanie Delaune and Lucca Hirschi. “A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols”. In: *Journal of Logical and Algebraic Methods in Programming* 87 (Feb. 2017), pp. 127–144. ISSN: 2352-2208. DOI: 10.1016/j.jlamp.2016.10.005. URL: http://dx.doi.org/10.1016/j.jlamp.2016.10.005.

[113] Stéphanie Delaune, Steve Kremer, and Olivier Pereira. *Simulation based security in the applied pi calculus*. Cryptology ePrint Archive, Report 2009/267. https://eprint.iacr.org/2009/267. 2009.

[114] Antoine Delignat-Lavaud et al. "Why Should I Trust Your Code?" In: *Commun. ACM* 67.1 (2024), pp. 68–76. DOI: 10.1145/3624578. URL: https://doi.org/10.1145/3624578.

[115] Yunjie Deng et al. "StrongBox: A GPU TEE on Arm Endpoints". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. Ed. by Heng Yin et al. ACM, 2022, pp. 769–783. ISBN: 978-1-4503-9450-5. DOI: 10.1145/3548606.3560627. URL: https://doi.org/10.1145/3548606.3560627.

[116] Yvo Desmedt and Jean-Jacques Quisquater. "Public-Key Systems Based on the Difficulty of Tampering (Is There a Difference Between DES and RSA?)" In: *Advances in Cryptology – CRYPTO'86*. Ed. by Andrew M. Odlyzko. Vol. 263. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1987, pp. 111–117. DOI: 10.1007/3-540-47721-7_9.

[117] Gobikrishna Dhanuskodi et al. "Creating the First Confidential GPUs: The team at NVIDIA brings confidentiality and integrity to user code and data for accelerated computing." In: *Queue* 21.4 (Aug. 2023), pp. 68–93. ISSN: 1542-7749. DOI: 10.1145/3623393.3623391. URL: http://dx.doi.org/10.1145/3623393.3623391.

[118] Diego Gambetta. "Can We Trust Trust?" In: (), pp. 213–237.

[119] Baltasar Dinis, Peter Druschel, and Rodrigo Rodrigues. "RR: A Fault Model for Efficient TEE Replication". In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/rr-a-fault-model-for-efficient-tee-replication/.

[120] Cory Doctorow. "The coming war on general computation". 28th Chaos Communication Congress. 2011. URL: https://media.ccc.de/v/28c3-4848-en-the_coming_war_on_general_computation.

[121] Natnatee Dokmai et al. "Privacy-preserving genotype imputation in a trusted execution environment". In: *Cell Systems* 12.10 (Oct. 2021), 983–993.e7. ISSN:

2405-4712. DOI: 10.1016/j.cels.2021.08.001. URL: http://dx.doi.org/10.1016/j.cels.2021.08.001.

[122] Felix Dörre, Jeremias Mechler, and Jörn Müller-Quade. "Practically Efficient Private Set Intersection from Trusted Hardware with Side-Channels". In: *Advances in Cryptology – ASIACRYPT 2023, Part IV*. Ed. by Jian Guo and Ron Steinfeld. Vol. 14441. Lecture Notes in Computer Science. Guangzhou, China: Springer, Singapore, Singapore, Dec. 2023, pp. 268–301. DOI: 10.1007/978-981-99-8730-6_9.

[123] Jules Drean et al. "Citadel: Side-Channel-Resistant Enclaves with Secure Shared Memory on a Speculative Out-of-Order Processor". In: *CoRR* abs/2306.14882 (2023). DOI: 10.48550/ARXIV.2306.14882. arXiv: 2306.14882. URL: https://doi.org/10.48550/arXiv.2306.14882.

[124] Andreas Erwig et al. "CommiTEE : An Efficient and Secure Commit-Chain Protocol using TEEs". In: *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*. IEEE, 2023, pp. 429–448. ISBN: 978-1-6654-6512-0. DOI: 10.1109/EuroSP57164.2023.00033. URL: https://doi.org/10.1109/EuroSP57164.2023.00033.

[125] Dengguo Feng et al. "Survey of research on confidential computing". In: *IET Communications* (2024).

[126] Andrew Ferraiuolo et al. "Komodo: Using verification to disentangle secure-enclave hardware from software". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. ACM, Oct. 2017. DOI: 10.1145/3132747.3132782. URL: http://dx.doi.org/10.1145/3132747.3132782.

[127] Ben Fisch et al. "IRON: Functional Encryption using Intel SGX". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 765–782. DOI: 10.1145/3133956.3134106.

[128] Ben A. Fisch et al. *Iron: Functional Encryption using Intel SGX*. Cryptology ePrint Archive, Report 2016/1071. https://eprint.iacr.org/2016/1071. 2016.

[129] Anthony C. J. Fox et al. "A Verification Methodology for the Arm® Confidential Computing Architecture: From a Secure Specification to Safe Implementations". In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA1 (Apr. 2023), pp. 376–405. ISSN: 2475-1421. DOI: 10.1145/3586040. URL: http://dx.doi.org/10.1145/3586040.

[130] Tommaso Frassetto et al. "POSE: Practical Off-chain Smart Contract Execution". In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/pose-practical-off-chain-smart-contract-execution/.

[131] Nicholas R.J. Frick et al. "The perceived surveillance of conversations through smart devices". In: *Electronic Commerce Research and Applications* 47 (May 2021), p. 101046. ISSN: 1567-4223. DOI: 10.1016/j.elerap.2021.101046. URL: http://dx.doi.org/10.1016/j.elerap.2021.101046.

[132] Georg Fuchsbauer, Antoine Plouviez, and Yannick Seurin. "Blind Schnorr Signatures and Signed ElGamal Encryption in the Algebraic Group Model". In: *Advances in Cryptology – EUROCRYPT 2020, Part II*. Ed. by Anne Canteaut and Yuval Ishai. Vol. 12106. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Cham, Switzerland, May 2020, pp. 63–95. DOI: 10.1007/978-3-030-45724-2_3.

[133] Francis Fukuyama. *Trust: The social virtues and the creation of prosperity*. Simon and Schuster, 1996.

[134] Sivanarayana Gaddam et al. "How to Design Fair Protocols in the Multi-Blockchain Setting". In: *IACR Cryptol. ePrint Arch.* (2023), p. 762. URL: https://eprint.iacr.org/2023/762.

[135] Sivanarayana Gaddam et al. "LucidiTEE: Scalable Policy-Based Multiparty Computation with Fairness". In: *CANS 23: 22th International Conference on Cryptology and Network Security*. Ed. by Jing Deng, Vladimir Kolesnikov, and Alexander A. Schwarzmann. Vol. 14342. Lecture Notes in Computer Science. Augusta, GA, USA: Springer, Singapore, Singapore, Oct. 2023, pp. 343–367. DOI: 10.1007/978-981-99-7563-1_16.

[136] Pranav Gaddamadugu. “Formally Verifying Trusted Execution Environments with UCLID5”. MA thesis. EECS Department, University of California, Berkeley, Aug. 2021. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-200.html`.

[137] Mingyuan Gao, Hung Dang, and Ee-Chien Chang. “TEEKAP: Self-Expiring Data Capsule using Trusted Execution Environment”. In: *ACSAC ’21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 2021, pp. 235–247. ISBN: 978-1-4503-8579-4. DOI: `10.1145/3485832.3485919`. URL: `https://doi.org/10.1145/3485832.3485919`.

[138] Cesare Garlati and Sandro Pinto. “A Clean Slate Approach to Linux Security RISC-V Enclaves”. In: *Embedded World*. Feb. 2020.

[139] R.E. Georgsen and G. Myrdahl Køien. “Serious Games with SysML: Gamifying Threat Modelling in a Small Business Setting”. In: *INCOSE International Symposium* 32.S2 (July 2022), pp. 119–132. ISSN: 2334-5837. DOI: `10.1002/iis2.12902`. URL: `http://dx.doi.org/10.1002/iis2.12902`.

[140] Santosh Ghosh et al. *A >100 Gbps Inline AES-GCM Hardware Engine and Protected DMA Transfers between SGX Enclave and FPGA Accelerator Device*. Cryptology ePrint Archive, Report 2020/178. `https://eprint.iacr.org/2020/178`. 2020.

[141] Oded Goldreich. “On Expected Probabilistic Polynomial-Time Adversaries: A Suggestion for Restricted Definitions and Their Benefits”. In: *TCC 2007: 4th Theory of Cryptography Conference*. Ed. by Salil P. Vadhan. Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer, Berlin, Heidelberg, Germany, Feb. 2007, pp. 174–193. DOI: `10.1007/978-3-540-70936-7_10`.

[142] Shafi Goldwasser and Yael Tauman Kalai. *Cryptographic Assumptions: A Position Paper*. Cryptology ePrint Archive, Report 2015/907. `https://eprint.iacr.org/2015/907`. 2015.

[143] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM Journal on Computing* 18.1 (1989), pp. 186–208.

[144] Shafi Goldwasser et al. “Multi-input Functional Encryption”. In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Copenhagen, Denmark: Springer, Berlin, Heidelberg, Germany, May 2014, pp. 578–602. DOI: 10.1007/978-3-642-55220-5_32.

[145] Owen Le Gonidec et al. *Do Not Trust Power Management: Challenges and Hints for Securing Future Trusted Execution Environments*. 2024. arXiv: 2405.15537v1 [cs.CR].

[146] Vipul Goyal et al. “Functional Encryption for Randomized Functionalities”. In: *TCC 2015: 12th Theory of Cryptography Conference, Part II*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9015. Lecture Notes in Computer Science. Warsaw, Poland: Springer, Berlin, Heidelberg, Germany, Mar. 2015, pp. 325–351. DOI: 10.1007/978-3-662-46497-7_13.

[147] Franz Gregor et al. “Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders”. In: *CoRR* abs/2003.14099 (2020). arXiv: 2003.14099. URL: https://arxiv.org/abs/2003.14099.

[148] Michele Grisafi et al. “PISTIS: Trusted Computing Architecture for Low-end Embedded Systems”. In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 3843–3860.

[149] Daniel Günther et al. *PEM: Privacy-preserving Epidemiological Modeling*. Cryptology ePrint Archive, Report 2020/1546. https://eprint.iacr.org/2020/1546. 2020.

[150] Suyash Gupta et al. “Dissecting BFT Consensus: In Trusted Components we Trust!” In: *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. Ed. by Giuseppe Antonio Di Luna et al. ACM, 2023, pp. 521–539. ISBN: 978-1-4503-9487-1. DOI: 10.1145/3552326.3587455. URL: https://doi.org/10.1145/3552326.3587455.

[151] Yaron Gvili. *Security Analysis of the COVID-19 Contact Tracing Specifications by Apple Inc. and Google Inc.* Cryptology ePrint Archive, Report 2020/428. https://eprint.iacr.org/2020/428. 2020.

[152] Farkhondeh Hassandoust, Saeed Akhlaghpour, and Allen C Johnston. "Individuals' privacy concerns and adoption of contact tracing mobile applications in a pandemic: A situational privacy calculus perspective". In: *Journal of the American Medical Informatics Association* 28.3 (Nov. 2020), pp. 463–471. ISSN: 1527-974X. eprint: `https://academic.oup.com/jamia/article-pdf/28/3/463/36428657/ocaa240.pdf`. URL: `https://doi.org/10.1093/jamia/ocaa240`.

[153] Dennis Hofheinz and Victor Shoup. *GNUC: A New Universal Composability Framework*. Cryptology ePrint Archive, Report 2011/303. `https://eprint.iacr.org/2011/303`. 2011.

[154] Andrew 'bunnie' Huang. *On Contact Tracing and Hardware Tokens*. 2020. URL: `https://www.bunniestudios.com/blog/?p=5820`.

[155] Haoyang Huang et al. "SoK: A Comparison Study of Arm TrustZone and CCA". In: (2024).

[156] Russell Impagliazzo. "A Personal View of Average-Case Complexity". In: *Proceedings of the Tenth Annual Structure in Complexity Theory Conference, Minneapolis, Minnesota, USA, June 19-22, 1995*. IEEE Computer Society, 1995, pp. 134–147. ISBN: 0-8186-7052-5. DOI: `10.1109/SCT.1995.514853`. URL: `https://doi.org/10.1109/SCT.1995.514853`.

[157] Charlie Jacomme, Steve Kremer, and Guillaume Scerri. *Symbolic Models for Isolated Execution Environments*. Cryptology ePrint Archive, Report 2017/070. `https://eprint.iacr.org/2017/070`. 2017.

[158] Sashidhar Jakkamsetti, Zeyu Liu, and Varun Madathil. "Scalable Private Signaling". In: *IACR Cryptol. ePrint Arch.* (2023), p. 572. URL: `https://eprint.iacr.org/2023/572`.

[159] Insu Jang et al. "Heterogeneous Isolated Execution for Commodity GPUs". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Apr. 2019, nil. DOI: `10.1145/3297858.3304021`. URL: `https://doi.org/10.1145/3297858.3304021`.

[160] Mohit Kumar Jangid et al. "Towards Formal Verification of State Continuity for Enclave Programs". In: *USENIX Security 2021: 30th USENIX Security*

Symposium. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 573–590.

[161] Jianyu Jiang et al. "CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment". In: *55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022*. IEEE, 2022, pp. 124–143. ISBN: 978-1-6654-6272-3. DOI: 10.1109/MICRO56248.2022.00019. URL: https://doi.org/10.1109/MICRO56248.2022.00019.

[162] Rüdiger Kapitza et al. "CheapBFT: resource-efficient byzantine fault tolerance". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 295–308. ISBN: 9781450312233. DOI: 10.1145/2168836.2168866. URL: https://doi.org/10.1145/2168836.2168866.

[163] Gabriel Kaptchuk, Matthew Green, and Ian Miers. "Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers". In: *ISOC Network and Distributed System Security Symposium – NDSS 2019*. San Diego, CA, USA: The Internet Society, Feb. 2019. DOI: 10.14722/ndss.2019.23060.

[164] Jonathan Katz. "Universally Composable Multi-party Computation Using Tamper-Proof Hardware". In: *Advances in Cryptology – EUROCRYPT 2007*. Ed. by Moni Naor. Vol. 4515. Lecture Notes in Computer Science. Barcelona, Spain: Springer, Berlin, Heidelberg, Germany, May 2007, pp. 115–128. DOI: 10.1007/978-3-540-72540-4_7.

[165] Jonathan Katz and Yehuda Lindell. "Handling Expected Polynomial-Time Strategies in Simulation-Based Security Proofs". In: *TCC 2005: 2nd Theory of Cryptography Conference*. Ed. by Joe Kilian. Vol. 3378. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2005, pp. 128–149. DOI: 10.1007/978-3-540-30576-7_8.

[166] Mahimna Kelkar et al. *Complete Knowledge: Preventing Encumbrance of Cryptographic Secrets*. Cryptology ePrint Archive, Report 2023/044. https://eprint.iacr.org/2023/044. 2023.

[167] Lina M Khan. "Sources of tech platform power". In: *Geo. L. Tech. Rev.* 2 (2017), p. 325.

[168] Gerwin Klein et al. “seL4: Formal verification of an OS kernel”. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[169] Ilan Komargodski, Gil Segev, and Eylon Yogev. “Functional Encryption for Randomized Functionalities in the Private-Key Setting from Minimal Assumptions”. In: *TCC 2015: 12th Theory of Cryptography Conference, Part II*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9015. Lecture Notes in Computer Science. Warsaw, Poland: Springer, Berlin, Heidelberg, Germany, Mar. 2015, pp. 352–377. DOI: `10.1007/978-3-662-46497-7_14`.

[170] Markus G. Kuhn. *The TrustNo 1 Cryptoprocessor Concept*. Tech. rep. University of Cambridge, 1997. URL: `https://www.cl.cam.ac.uk/~mgk25/trustno1.pdf`.

[171] Rashna Kumar et al. “Each at its Own Pace: Third-Party Dependency and Centralization Around the World”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 7.1 (Feb. 2023), pp. 1–29. ISSN: 2476-1249. DOI: `10.1145/3579437`. URL: `http://dx.doi.org/10.1145/3579437`.

[172] Ralf Küsters and Daniel Rausch. “A Framework for Universally Composable Diffie-Hellman Key Exchange”. In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2017, pp. 881–900. DOI: `10.1109/SP.2017.63`.

[173] Ralf Küsters and Max Tuengerthal. *The IITM Model: a Simple and Expressive Model for Universal Composability*. Cryptology ePrint Archive, Report 2013/025. `https://eprint.iacr.org/2013/025`. 2013.

[174] Benjamin Laufer and Helen Nissenbaum. “Algorithmic displacement of social trust”. In: *Knight First Amendment Institute at Columbia University* (2023).

[175] Dayeol Lee et al. “Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing”. In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin et al. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 1871–1885. DOI: `10.1145/3548606.3560595`.

[176] Dayeol Lee et al. “Keystone: An Open Framework for Architecting Trusted Execution Environments”. In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Com-

puting Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387532. URL: https://doi.org/10.1145/3342195.3387532.

[177] Barry M Leiner et al. “A brief history of the Internet”. In: *ACM SIGCOMM computer communication review* 39.5 (2009), pp. 22–31.

[178] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. “Verifying Linearizability of Intel® Software Guard Extensions”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2015, pp. 144–160. ISBN: 9783319216683. DOI: 10.1007/978-3-319-21668-3_9. URL: http://dx.doi.org/10.1007/978-3-319-21668-3_9.

[179] Dave Levin et al. “TrInc: Small Trusted Hardware for Large Distributed Systems.” In: *NSDI*. Vol. 9. 2009, pp. 1–14.

[180] Steven Levy. *Crypto: How the code rebels beat the government–saving privacy in the digital age*. Penguin, 2001.

[181] Mengyuan Li et al. “SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments”. In: *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (2024).

[182] Ming Li et al. *IvyCross: A Trustworthy and Privacy-preserving Framework for Blockchain Interoperability*. Cryptology ePrint Archive, Report 2021/1244. https://eprint.iacr.org/2021/1244. 2021.

[183] Mingyu Li, Yubin Xia, and Haibo Chen. “Confidential Serverless Made Efficient with Plug-In Enclaves”. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 306–318. DOI: 10.1109/ISCA52012.2021.00032.

[184] Mingyu (Maxul) Li. *Awesome SGX Open Source Projects*. GitHub (https://github.com/Maxul/Awesome-SGX-Open-Source). 2019. (Visited on 05/17/2024).

[185] Jinghui Liao et al. “Speedster: An Efficient Multi-party State Channel via Enclaves”. In: *ASIACCS 22: 17th ACM Symposium on Information, Computer and Communications Security*. Ed. by Yuji Suga et al. Nagasaki, Japan: ACM Press, May 2022, pp. 637–651. DOI: 10.1145/3488932.3523259.

[186] Joshua Lind et al. “Teechain: a secure payment network with asynchronous blockchain access”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 63–79. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359627. URL: https://doi.org/10.1145/3341301.3359627.

[187] Yehuda Lindell. “A Simpler Construction of CCA2-Secure Public-Key Encryption under General Assumptions”. In: *Advances in Cryptology – EUROCRYPT 2003*. Ed. by Eli Biham. Vol. 2656. Lecture Notes in Computer Science. Warsaw, Poland: Springer, Berlin, Heidelberg, Germany, May 2003, pp. 241–254. DOI: 10.1007/3-540-39200-9_15.

[188] Yehuda Lindell. *General Composition and Universal Composability in Secure Multiparty Computation*. Cryptology ePrint Archive, Report 2003/141. https://eprint.iacr.org/2003/141. 2003.

[189] Yehuda Lindell. *How To Simulate It - A Tutorial on the Simulation Proof Technique*. Cryptology ePrint Archive, Report 2016/046. https://eprint.iacr.org/2016/046. 2016.

[190] Yibiao Lu et al. “Correlated Randomness Teleportation via Semi-trusted Hardware - Enabling Silent Multi-party Computation”. In: *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part II*. Ed. by Elisa Bertino, Haya Shulman, and Michael Waidner. Vol. 12973. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Cham, Switzerland, Oct. 2021, pp. 699–720. DOI: 10.1007/978-3-030-88428-4_34.

[191] Wouter Lueks et al. “CrowdNotifier: Decentralized Privacy-Preserving Presence Tracing”. In: *Proceedings on Privacy Enhancing Technologies* 2021.4 (Oct. 2021), pp. 350–368. DOI: 10.2478/popets-2021-0074.

[192] Niklas Luhmann et al. *Trust and Power : two works by Niklas Luhmann / with an introduction by Gianfranco Poggi*. eng. Chichester: Wiley, 1979. ISBN: 0471997587.

[193] Joshua Lund. *Technology Preview for secure value recovery*. https://signal.org/blog/secure-value-recovery. Dec. 2019.

[194] Yao Ma et al. “QEnclave - A practical solution for secure quantum cloud computing”. In: *CoRR* abs/2109.02952 (2021). arXiv: `2109.02952`. URL: `https://arxiv.org/abs/2109.02952`.

[195] Varun Madathil et al. “Private Signaling”. In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 3309–3326.

[196] Wenze Mao, Peng Jiang, and Liehuang Zhu. “BTAA: Blockchain and TEE-Assisted Authentication for IoT Systems”. In: *IEEE Internet of Things Journal* 10.14 (July 2023), pp. 12603–12615. ISSN: 2372-2541. DOI: `10.1109/jiot.2023.3252565`. URL: `http://dx.doi.org/10.1109/jiot.2023.3252565`.

[197] André Martin et al. “ADAM-CS: Advanced Asynchronous Monotonic Counter Service”. In: *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 2021, pp. 426–437. ISBN: 978-1-6654-3572-7. DOI: `10.1109/DSN48987.2021.00053`. URL: `https://doi.org/10.1109/DSN48987.2021.00053`.

[198] Tania Martin et al. “Demystifying Covid-19 Digital Contact Tracing: a Survey on Frameworks and Mobile Apps”. In: *CoRR* (2020).

[199] Carla Mascia, Massimiliano Sala, and Irene Villa. “A survey on Functional Encryption”. In: *CoRR* abs/2106.06306 (2021). arXiv: `2106.06306`. URL: `https://arxiv.org/abs/2106.06306`.

[200] Sinisa Matetic et al. “ROTE: Rollback Protection for Trusted Execution”. In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 1289–1306.

[201] Sinisa Matetic et al. *ROTE: Rollback Protection for Trusted Execution*. Cryptology ePrint Archive, Report 2017/048. `https://eprint.iacr.org/2017/048`. 2017.

[202] Christian Matt and Ueli Maurer. *A Definitional Framework for Functional Encryption*. Cryptology ePrint Archive, Report 2013/559. `https://eprint.iacr.org/2013/559`. 2013.

[203] Christian Matt and Ueli Maurer. “A Definitional Framework for Functional Encryption”. In: *CSF 2015: IEEE 28th Computer Security Foundations Symposium*. Ed. by Cedric Fournet and Michael Hicks. Verona, Italy: IEEE Computer Society Press, July 2015, pp. 217–231. DOI: 10.1109/CSF.2015.22.

[204] Ueli Maurer. “Constructive Cryptography - A New Paradigm for Security Definitions and Proofs”. In: *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*. Ed. by Sebastian Mödersheim and Catuscia Palamidessi. Vol. 6993. Lecture Notes in Computer Science. Springer, 2011, pp. 33–56. ISBN: 978-3-642-27374-2. DOI: 10.1007/978-3-642-27375-9_3. URL: https://doi.org/10.1007/978-3-642-27375-9%5C_3.

[205] Ueli Maurer. “Constructive Cryptography - A Primer (Invited Paper)”. In: *FC 2010: 14th International Conference on Financial Cryptography and Data Security*. Ed. by Radu Sion. Vol. 6052. Lecture Notes in Computer Science. Tenerife, Canary Islands, Spain: Springer, Berlin, Heidelberg, Germany, Jan. 2010, p. 1. DOI: 10.1007/978-3-642-14577-3_1.

[206] Ueli Maurer and Renato Renner. “Abstract Cryptography”. In: *ICS 2011: 2nd Innovations in Computer Science*. Ed. by Bernard Chazelle. Tsinghua University, Beijing, China: Tsinghua University Press, Jan. 2011, pp. 1–21.

[207] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. “Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology”. In: *TCC 2004: 1st Theory of Cryptography Conference*. Ed. by Moni Naor. Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2004, pp. 21–39. DOI: 10.1007/978-3-540-24638-1_2.

[208] Frank McKeen et al. “Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave”. In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*. ACM, 2016, 10:1–10:9. ISBN: 978-1-4503-4769-3. DOI: 10.1145/2948618.2954331. URL: https://doi.org/10.1145/2948618.2954331.

[209] Microsoft. *Windows 11 System Requirements*. https://support.microsoft.com/en-us/windows/windows-11-system-requirements-86c11283-ea52-4782-9efd-7674389a7ba3. Aug. 2021.

[210] Koichi Moriyama and Akira Otsuka. "Permissionless Blockchain-Based Sybil-Resistant Self-Sovereign Identity Utilizing Attested Execution Secure Processors". In: *IEEE International Conference on Blockchain, Blockchain 2022, Espoo, Finland, August 22-25, 2022*. IEEE, 2022, pp. 1–10. ISBN: 978-1-6654-6104-7. DOI: `10.1109/Blockchain55522.2022.00012`. URL: `https://doi.org/10.1109/Blockchain55522.2022.00012`.

[211] Giovane C. M. Moura et al. "Clouding up the Internet: how centralized is DNS traffic becoming?" In: *Proceedings of the ACM Internet Measurement Conference*. IMC '20. ACM, Oct. 2020. DOI: `10.1145/3419394.3423625`. URL: `http://dx.doi.org/10.1145/3419394.3423625`.

[212] Randall Munroe. *Security*. `https://xkcd.com/538/`. XKCD. 2009.

[213] Arvind Narayanan. "What Happened to the Crypto Dream?, Part 1". In: *IEEE Security & Privacy* 11.2 (2013), pp. 75–76. DOI: `10.1109/MSP.2013.45`.

[214] Arvind Narayanan et al. "Location Privacy via Private Proximity Testing". In: *ISOC Network and Distributed System Security Symposium – NDSS 2011*. San Diego, CA, USA: The Internet Society, Feb. 2011.

[215] Pascal Nasahl et al. "HECTOR-V: A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment". In: *ASIACCS 21: 16th ACM Symposium on Information, Computer and Communications Security*. Ed. by Jiannong Cao et al. Virtual Event, Hong Kong: ACM Press, June 2021, pp. 187–199. DOI: `10.1145/3433210.3453112`.

[216] John Naughton. "The evolution of the Internet: from military experiment to General Purpose Technology". In: *Journal of Cyber Policy* 1.1 (Jan. 2016), pp. 5–28. ISSN: 2373-8898. DOI: `10.1080/23738871.2016.1157619`. URL: `http://dx.doi.org/10.1080/23738871.2016.1157619`.

[217] Kartik Nayak et al. "HOP: Hardware makes Obfuscation Practical". In: *ISOC Network and Distributed System Security Symposium – NDSS 2017*. San Diego, CA, USA: The Internet Society, Feb. 2017. DOI: `10.14722/ndss.2017.23349`.

[218] *ISOC Network and Distributed System Security Symposium – NDSS 2019*. San Diego, CA, USA: The Internet Society, Feb. 2019.

[219] *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss2023/.

[220] Jianyu Niu et al. "NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin et al. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 2385–2399. DOI: 10.1145/3548606.3560620.

[221] Job Noorman et al. "Sancus 2.0: A Low-Cost Security Architecture for IoT Devices". In: *ACM Trans. Priv. Secur.* 20.3 (2017), 7:1–7:33. DOI: 10.1145/3079763. URL: https://doi.org/10.1145/3079763.

[222] Hyunyoung Oh et al. "MeetGo: A Trusted Execution Environment for Remote Applications on FPGA". In: *IEEE Access* 9 (2021), pp. 51313–51324. DOI: 10.1109/ACCESS.2021.3069223. URL: https://doi.org/10.1109/ACCESS.2021.3069223.

[223] H. Orman. "The Morris worm: a fifteen-year perspective". In: *IEEE Security & Privacy* 1.5 (2003), pp. 35–43. DOI: 10.1109/MSECP.2003.1236233.

[224] Chris Orsini, Alessandra Scafuro, and Tanner Verber. *How to Recover a Cryptographic Secret From the Cloud*. Cryptology ePrint Archive, Paper 2023/1308. https://eprint.iacr.org/2023/1308. 2023. URL: https://eprint.iacr.org/2023/1308.

[225] Arttu Paju et al. "SoK: A Systematic Review of TEE Usage for Developing Trusted Applications". In: *Proceedings of the 18th International Conference on Availability, Reliability and Security*. ARES 2023. ACM, Aug. 2023. DOI: 10.1145/3600160.3600169. URL: http://dx.doi.org/10.1145/3600160.3600169.

[226] Joongun Park et al. "Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020. DOI: 10.1109/isca45697.2020.00069. URL: http://dx.doi.org/10.1109/ISCA45697.2020.00069.

[227] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. “Bootstrapping Trust in Commodity Computers”. In: *2010 IEEE Symposium on Security and Privacy*. Berkeley/Oakland, CA, USA: IEEE Computer Society Press, May 2010, pp. 414–429. DOI: 10.1109/SP.2010.32.

[228] Bryan Parno et al. “Memoir: Practical State Continuity for Protected Modules”. In: *2011 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 2011, pp. 379–394. DOI: 10.1109/SP.2011.38.

[229] Rafael Pass, Elaine Shi, and Florian Tramer. *Formal Abstractions for Attested Execution Secure Processors*. Cryptology ePrint Archive, Report 2016/1027. https://eprint.iacr.org/2016/1027. 2016.

[230] Rafael Pass, Elaine Shi, and Florian Tramèr. “Formal Abstractions for Attested Execution Secure Processors”. In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Paris, France: Springer, Cham, Switzerland, Apr. 2017, pp. 260–289. DOI: 10.1007/978-3-319-56620-7_10.

[231] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. “Exploring Widevine for Fun and Profit”. In: *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, May 2022. DOI: 10.1109/spw54247.2022.9833867. URL: http://dx.doi.org/10.1109/SPW54247.2022.9833867.

[232] Sérgio Pereira et al. “Towards a Trusted Execution Environment via Reconfigurable FPGA”. In: *CoRR* abs/2107.03781 (2021). arXiv: 2107.03781. URL: https://arxiv.org/abs/2107.03781.

[233] Birgit Pfitzmann and Michael Waidner. *A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission*. Cryptology ePrint Archive, Report 2000/066. https://eprint.iacr.org/2000/066. 2000.

[234] Ines Pinto Gouveia et al. “To verify or tolerate, that’s the question”. In: *Program Analysis and Verification on Trusted Platforms (PAVeTrust) Workshop (co-located with ACSAC21)*. 2021.

[235] Ajith Ramanathan et al. “Probabilistic Bisimulation and Equivalence for Security Analysis of Network Protocols”. In: *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004, Held*

as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Ed. by Igor Walukiewicz. Vol. 2987. Lecture Notes in Computer Science. Springer, 2004, pp. 468–483. ISBN: 3-540-21298-1. DOI: 10.1007/978-3-540-24727-2_33. URL: https://doi.org/10.1007/978-3-540-24727-2%5C_33.

[236] Leonie Reichert, Samuel Brack, and Björn Scheuermann. *Ovid: Message-based Automatic Contact Tracing*. Cryptology ePrint Archive, Report 2020/1462. https://eprint.iacr.org/2020/1462. 2020.

[237] Phillip Rogaway. *The Moral Character of Cryptographic Work*. Cryptology ePrint Archive, Report 2015/1162. https://eprint.iacr.org/2015/1162. 2015.

[238] Mark Rosso, ABM Nasir, and Mohsen Farhadloo. “Chilling effects and the stock market response to the Snowden revelations”. In: *New Media & Society* 22.11 (Oct. 2020), pp. 1976–1995. ISSN: 1461-7315. DOI: 10.1177/1461444820924619. URL: http://dx.doi.org/10.1177/1461444820924619.

[239] Xiaoyu Ruan. *Platform Embedded Security Technology Revealed*. Apress, 2014. ISBN: 9781430265726. DOI: 10.1007/978-1-4302-6572-6. URL: http://dx.doi.org/10.1007/978-1-4302-6572-6.

[240] Mark Russinovich et al. “CCF: A framework for building confidential verifiable replicated services”. In: *Technical Report MSR-TR-2019-16, Microsoft, Tech. Rep.* (2019).

[241] Joanna Rutkowska. *Intel x86 considered harmful*. Tech. rep. the Invisible Things Lab, 2015.

[242] Amit Sahai and Brent R. Waters. “Fuzzy Identity-Based Encryption”. In: *Advances in Cryptology – EUROCRYPT 2005*. Ed. by Ronald Cramer. Vol. 3494. Lecture Notes in Computer Science. Aarhus, Denmark: Springer, Berlin, Heidelberg, Germany, May 2005, pp. 457–473. DOI: 10.1007/11426639_27.

[243] Ammar S. Salman and Wenliang Du. “Securing Mobile Systems GPS and Camera Functions Using TrustZone Framework”. In: *Intelligent Computing*. Springer International Publishing, 2021, pp. 868–884. ISBN: 9783030801298. DOI: 10.1007/978-3-030-80129-8_58. URL: http://dx.doi.org/10.1007/978-3-030-80129-8_58.

[244] Muhammad Usama Sardar, Rasha Faqeh, and Christof Fetzer. "Formal Foundations for Intel SGX Data Center Attestation Primitives". In: *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*. Ed. by Shang-Wei Lin, Zhe Hou, and Brendan Mahoney. Vol. 12531. Lecture Notes in Computer Science. Springer, 2020, pp. 268–283. ISBN: 978-3-030-63405-6. DOI: 10.1007/978-3-030-63406-3_16. URL: https://doi.org/10.1007/978-3-030-63406-3%5C_16.

[245] Muhammad Usama Sardar and Christof Fetzer. "Confidential computing and related technologies: a critical review". In: *Cybersecur.* 6.1 (2023), p. 10. DOI: 10.1186/S42400-023-00144-1. URL: https://doi.org/10.1186/s42400-023-00144-1.

[246] Muhammad Usama Sardar and Christof Fetzer. "Formal Foundations for SCONE Attestation". In: *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Supplemental Volume, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 2022, pp. 31–32. ISBN: 978-1-6654-0260-6. DOI: 10.1109/DSN-S54099.2022.00020. URL: https://doi.org/10.1109/DSN-S54099.2022.00020.

[247] Muhammad Usama Sardar, Thomas Fossati, and Simon Frost. *SoK: Attestation in confidential computing*. Researchgate.net pre-print. https://www.researchgate.net/profile/Muhammad-Sardar-6/publication/367284929_SoK_Attestation_in_Confidential_Computing/links/63ca778ad9fb5967c2ef3363/SoK-Attestation-in-Confidential-Computing.pdf. 2022.

[248] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. "Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification". In: *IEEE Access* 9 (2021), pp. 83067–83079. DOI: 10.1109/ACCESS.2021.3087421. URL: https://doi.org/10.1109/ACCESS.2021.3087421.

[249] Muhammad Usama Sardar, Do Le Quoc, and Christof Fetzer. "Towards Formalization of Enhanced Privacy ID (EPID)-based Remote Attestation in Intel SGX". In: *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 2020, pp. 604–607. ISBN: 978-1-7281-9535-3. DOI: 10.1109/DSD51259.2020.00099. URL: https://doi.org/10.1109/DSD51259.2020.00099.

[250] Muhammad Usama Sardar et al. “Formal Specification and Verification of Architecturally-Defined Attestation Mechanisms in Arm CCA and Intel TDX”. In: *IEEE Access* 12 (2024), pp. 361–381. DOI: 10.1109/ACCESS.2023.3346501. URL: https://doi.org/10.1109/ACCESS.2023.3346501.

[251] Stephan van Schaik et al. *SoK: SGX.Fail: How Stuff Get eXposed*. https://sgx.fail. 2022.

[252] Moritz Schneider et al. “Composite Enclaves: Towards Disaggregated Trusted Execution”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2022.1 (2022), pp. 630–656. DOI: 10.46586/tches.v2022.i1.630-656.

[253] Moritz Schneider et al. *SoK: Hardware-supported Trusted Execution Environments*. 2022. arXiv: 2205.12742v1 [cs.CR].

[254] Claus-Peter Schnorr and Markus Jakobsson. “Security of Signed ElGamal Encryption”. In: *Advances in Cryptology – ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Kyoto, Japan: Springer, Berlin, Heidelberg, Germany, Dec. 2000, pp. 73–89. DOI: 10.1007/3-540-44448-3_7.

[255] Paul M Schwartz. “Privacy and the economics of personal health care information”. In: *Tex. L. Rev.* 76 (1997), p. 1.

[256] *Security requirements for cryptographic modules*. Apr. 2019. DOI: 10.6028/nist.fips.140-3. URL: http://dx.doi.org/10.6028/NIST.FIPS.140-3.

[257] Adi Shamir. “Identity-Based Cryptosystems and Signature Schemes”. In: *Advances in Cryptology – CRYPTO’84*. Ed. by G. R. Blakley and David Chaum. Vol. 196. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Berlin, Heidelberg, Germany, Aug. 1984, pp. 47–53. DOI: 10.1007/3-540-39568-7_5.

[258] Victor Shoup. *Sequences of games: a tool for taming complexity in security proofs*. Cryptology ePrint Archive, Report 2004/332. https://eprint.iacr.org/2004/332. 2004.

[259] Rohit Sinha et al. "A design and verification methodology for secure isolated regions". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 665–681. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908113. URL: https://doi.org/10.1145/2908080.2908113.

[260] Rohit Sinha et al. "Moat: Verifying Confidentiality of Enclave Programs". In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 2015, pp. 1169–1184. DOI: 10.1145/2810103.2813608.

[261] Adam Smith. *The Wealth of Nations*. eng. Harriman House Publishing, 2007. ISBN: 1-906659-87-7.

[262] Raoul Strackx, Bart Jacobs, and Frank Piessens. "ICE: a passive, high-speed, state-continuity scheme". In: *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*. Ed. by Charles N. Payne Jr. et al. ACM, 2014, pp. 106–115. ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664259. URL: https://doi.org/10.1145/2664243.2664259.

[263] Raoul Strackx and Frank Piessens. "Ariadne: A Minimal Approach to State Continuity". In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 875–892.

[264] Siegfried Streufert. "Trust. A Mechanism for the Reduction of Social Complexity". eng. In: *Philosophy and history* 1.2 (1968), pp. 183–184. ISSN: 0016-884X.

[265] Pramod Subramanyan et al. "A Formal Foundation for Secure Remote Execution of Enclaves". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham et al. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 2435–2450. DOI: 10.1145/3133956.3134098.

[266] Haiyong Sun and Hang Lei. “A Design and Verification Methodology for a TrustZone Trusted Execution Environment”. In: *IEEE Access* 8 (2020), pp. 33870–33883. ISSN: 2169-3536. DOI: 10.1109/access.2020.2974487. URL: http://dx.doi.org/10.1109/ACCESS.2020.2974487.

[267] Intel Support. *Unable to find Alternatives to Monotonic Counter Application Programming Interfaces (APIs) in Intel® Software Guard Extensions (Intel® SGX) for Linux* to Prevent Sealing Rollback Attacks.* https://www.intel.com/content/www/us/en/support/articles/000057968/software/intel-security-products.html. May 2021.

[268] Tatsuya Suzuki et al. *Verifiable Functional Encryption using Intel SGX.* Cryptology ePrint Archive, Report 2020/1221. https://eprint.iacr.org/2020/1221. 2020.

[269] Taisei Takahashi, Taishi Higuchi, and Akira Otsuka. “VeloCash: Anonymous Decentralized Probabilistic Micropayments With Transferability”. In: *IEEE Access* 10 (2022), pp. 93701–93730. DOI: 10.1109/ACCESS.2022.3201071. URL: https://doi.org/10.1109/ACCESS.2022.3201071.

[270] Moni Naor, ed. *TCC 2004: 1st Theory of Cryptography Conference.* Vol. 2951. Lecture Notes in Computer Science. Cambridge, MA, USA: Springer, Berlin, Heidelberg, Germany, Feb. 2004.

[271] Salil P. Vadhan, ed. *TCC 2007: 4th Theory of Cryptography Conference.* Vol. 4392. Lecture Notes in Computer Science. Amsterdam, The Netherlands: Springer, Berlin, Heidelberg, Germany, Feb. 2007.

[272] Yevgeniy Dodis and Jesper Buus Nielsen, eds. *TCC 2015: 12th Theory of Cryptography Conference, Part II.* Vol. 9015. Lecture Notes in Computer Science. Warsaw, Poland: Springer, Berlin, Heidelberg, Germany, Mar. 2015.

[273] Compared to the Size of Economies The World’s Tech Giants. *Omri Wallach.* Ed. by Visual Capitalist. https://www.visualcapitalist.com/the-tech-giants-worth-compared-economies-countries/. 2021.

[274] Ken Thompson. “Reflections on Trusting Trust”. In: *Commun. ACM* 27.8 (1984), pp. 761–763. DOI: 10.1145/358198.358210. URL: https://doi.org/10.1145/358198.358210.

[275] Rafael Tonicelli, Bernardo Machado David, and Vinícius de Morais Alves. "Universally Composable Private Proximity Testing". In: *ProvSec 2011: 5th International Conference on Provable Security*. Ed. by Xavier Boyen and Xiaofeng Chen. Vol. 6980. Lecture Notes in Computer Science. Xi'an, China: Springer, Berlin, Heidelberg, Germany, Oct. 2011, pp. 222–239. DOI: 10.1007/978-3-642-24316-5_16.

[276] Florian Tramer et al. *Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge*. Cryptology ePrint Archive, Report 2016/635. https://eprint.iacr.org/2016/635. 2016.

[277] Florian Tramèr and Dan Boneh. "Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL: https://openreview.net/forum?id=rJVorjCcKQ.

[278] Florian Tramèr et al. "Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge". In: *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE. IEEE, 2017, pp. 19–34. ISBN: 978-1-5090-5762-7. DOI: 10.1109/EuroSP.2017.28. URL: https://doi.org/10.1109/EuroSP.2017.28.

[279] Ni Trieu et al. "Epione: Lightweight Contact Tracing with Strong Privacy". In: *IEEE Data Eng. Bull.* 43.2 (2020), pp. 95–107. URL: http://sites.computer.org/debull/A20june/p95.pdf.

[280] Carmela Troncoso et al. "Decentralized Privacy-Preserving Proximity Tracing". In: *IEEE Data Eng. Bull.* 43.2 (2020), pp. 36–66.

[281] Chia-che Tsai, Donald E. Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX". In: *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*. Ed. by Dilma Da Silva and Bryan Ford. USENIX Association, 2017, pp. 645–658. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[282] Hannes Tschofenig et al. *Trusted Execution Environment Provisioning (TEEP) Protocol*. Internet-Draft draft-ietf-teep-protocol-19. Work in Progress. Internet

Engineering Task Force, May 2024. 74 pp. URL: `https://datatracker.ietf.org/doc/draft-ietf-teep-protocol/19/`.

[283] Chenxu Wang University) et al. “CAGE: Complementing Arm CCA with GPU Extensions”. In: *Proceedings 2024 Network and Distributed System Security Symposium* (2024). URL: `https://api.semanticscholar.org/CorpusID:267621562`.

[284] Thorsten Holz and Stefan Savage, eds. *USENIX Security 2016: 25th USENIX Security Symposium*. Austin, TX, USA: USENIX Association, Aug. 2016.

[285] Kevin R. B. Butler and Kurt Thomas, eds. *USENIX Security 2022: 31st USENIX Security Symposium*. Boston, MA, USA: USENIX Association, Aug. 2022.

[286] Thomas Vanlaere. *Azure Confidential Computing: CoCo - Confidential Containers*. `https://thomasvanlaere.com/posts/2024/03/azure-confidential-computing-coco-confidential-containers/`. Mar. 2024.

[287] Serge Vaudenay. *Centralized or Decentralized? The Contact Tracing Dilemma*. Cryptology ePrint Archive, Report 2020/531. `https://eprint.iacr.org/2020/531`. 2020.

[288] Serge Vaudenay and Martin Vuagnoux. *The Dark Side of SwissCovid*. `https://lasec.epfl.ch/people/vaudenay/swisscovid`. 2020.

[289] Giuliana Santos Veronese et al. “Efficient Byzantine Fault-Tolerance”. In: *IEEE Transactions on Computers* 62.1 (Jan. 2013), pp. 16–30. ISSN: 0018-9340. DOI: `10.1109/tc.2011.221`. URL: `http://dx.doi.org/10.1109/TC.2011.221`.

[290] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. “Graviton: Trusted Execution Environments on GPUs”. In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 681–696. URL: `https://www.usenix.org/conference/osdi18/presentation/volos`.

[291] Kobe Vrancken and Frank Piessens. “Do we need consumer-side enclaved execution?” In: *Proceedings of the 5th Workshop on System Software for Trusted Execution (SysTEX ’22 Workshop), ACM, New York, NY, USA*. 2022.

[292] Ivana Vukotic, Vincent Rahli, and Paulo Esteves-Veríssimo. “Asphalion: trustworthy shielding against Byzantine faults”. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), pp. 1–32. ISSN: 2475-1421. DOI: `10.1145/3360564`. URL: `http://dx.doi.org/10.1145/3360564`.

[293] Margaret Urban Walker. *Moral repair: Reconstructing moral relations after wrongdoing*. Cambridge University Press, 2006.

[294] Weili Wang et al. “ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes”. In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin et al. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 2841–2855. DOI: `10.1145/3548606.3560639`.

[295] Carsten Weinhold et al. “Towards Modular Trusted Execution Environments”. In: *Proceedings of the 6th Workshop on System Software for Trusted Execution* (2023). URL: `https://api.semanticscholar.org/CorpusID:259234025`.

[296] Kevin Werbach. *The blockchain and the new architecture of trust*. Mit Press, 2018.

[297] Lucie White and Philippe van Basshuysen. “Privacy versus Public Health? A Reassessment of Centralised and Decentralised Digital Contact Tracing”. In: *Sci. Eng. Ethics* 27.2 (2021), p. 23.

[298] Newton C. Will and Carlos A. Maziero. “Intel Software Guard Extensions Applications: A Survey”. In: *ACM Computing Surveys* 55.14s (July 2023), pp. 1–38. ISSN: 1557-7341. DOI: `10.1145/3593021`. URL: `http://dx.doi.org/10.1145/3593021`.

[299] Rafal Wojtczuk and Joanna Rutkowska. “Attacking intel trusted execution technology”. In: *Black Hat DC* 2009 (2009), pp. 1–6.

[300] Pengfei Wu et al. “Exploring Dynamic Task Loading in SGX-Based Distributed Computing”. In: *IEEE Trans. Serv. Comput.* 16.1 (2023), pp. 288–301. DOI: `10.1109/TSC.2021.3123511`. URL: `https://doi.org/10.1109/TSC.2021.3123511`.

[301] Pengfei Wu et al. “ObliDC: An SGX-based Oblivious Distributed Computing Framework with Formal Proof”. In: *ASIACCS 19: 14th ACM Symposium on Information, Computer and Communications Security*. Ed. by Steven D. Galbraith et al. Auckland, New Zealand: ACM Press, July 2019, pp. 86–99. DOI: `10.1145/3321705.3329822`.

[302] Ke Xia et al. “SGX-FPGA: Trusted Execution Environment for CPU-FPGA Heterogeneous Architecture”. In: *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 2021, pp. 301–306. ISBN: 978-1-6654-3274-0. DOI: 10.1109/DAC18074.2021.9586207. URL: https://doi.org/10.1109/DAC18074.2021.9586207.

[303] Rongwu Xu et al. “Miso: Legacy-Compatible Privacy-Preserving Single Sign-On Using Trusted Execution Environments”. In: *CoRR* (2023). arXiv: 2305.06833 [cs.CR]. URL: http://arxiv.org/abs/2305.06833v2.

[304] Shiwei Xu et al. “A Symbolic Model for Systematically Analyzing TEE-Based Protocols”. In: *ICICS 20: 22nd International Conference on Information and Communication Security*. Ed. by Weizhi Meng et al. Vol. 11999. Lecture Notes in Computer Science. Copenhagen, Denmark: Springer, Cham, Switzerland, Aug. 2020, pp. 126–144. DOI: 10.1007/978-3-030-61078-4_8.

[305] Sravya Yandamuri et al. *Communication-Efficient BFT Protocols Using Small Trusted Hardware to Tolerate Minority Corruption*. Cryptology ePrint Archive, Report 2021/184. https://eprint.iacr.org/2021/184. 2021.

[306] Scott Yilek. “Resettable Public-Key Encryption: How to Encrypt on a Virtual Machine”. In: *Topics in Cryptology – CT-RSA 2010*. Ed. by Josef Pieprzyk. Vol. 5985. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Berlin, Heidelberg, Germany, Mar. 2010, pp. 41–56. DOI: 10.1007/978-3-642-11925-5_4.

[307] Jason Zhijingcheng Yu et al. “Elasticlave: An Efficient Memory Model for Enclaves”. In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 4111–4128.

[308] Shaza Zeitouni et al. “Trusted Configuration in Cloud FPGAs”. In: *29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2021, Orlando, FL, USA, May 9-12, 2021*. IEEE, 2021, pp. 233–241. ISBN: 978-1-6654-3555-0. DOI: 10.1109/FCCM51124.2021.00036. URL: https://doi.org/10.1109/FCCM51124.2021.00036.

[309] Fan Zhang et al. *Paralysis Proofs: Secure Access-Structure Updates for Cryptocurrencies and More*. Cryptology ePrint Archive, Report 2018/096. https://eprint.iacr.org/2018/096. 2018.

[310] Fan Zhang et al. “Town Crier: An Authenticated Data Feed for Smart Contracts”. In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl et al. Vienna, Austria: ACM Press, Oct. 2016, pp. 270–282. DOI: `10.1145/2976749.2978326`.

[311] Shixuan Zhao et al. “Reusable Enclaves for Confidential Serverless Computing”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*. Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 4015–4032. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan`.

[312] Xuyang Zhao et al. “Towards A Secure Joint Cloud With Confidential Computing”. In: *2022 IEEE International Conference on Joint Cloud Computing (JCC)* (Aug. 2022). DOI: `10.1109/jcc56315.2022.00019`. URL: `http://dx.doi.org/10.1109/jcc56315.2022.00019`.

[313] Jianping Zhu et al. “Enabling Rack-scale Confidential Computing using Heterogeneous Trusted Execution Environment”. In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 1450–1465. DOI: `10.1109/SP40000.2020.00054`.

[314] Shoshana Zuboff. *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*. eng. London: Profile, 2019. ISBN: 9781781256855.