# AGATE: Augmented Global Attested Trusted Execution in the Universal Composability framework

Lorenzo Martinico[1,2],Markulf Kohlweiss[1,2]

[1] University of Edinburgh, UK, `firstname.lastname@ed.ac.uk`
[2] IOG, UK

**Abstract.** A Trusted Execution Environment (TEE) is a new type of security technology, implemented by CPU manufacturers, which guarantees integrity and confidentiality on a restricted execution environment to any remote verifier. TEEs are deployed on various consumer and commercial hardwareplatforms, and have been widely adopted as a component in the design of cryptographic protocols both theoretical and practical.
Within the provable security community, the use of TEEs as a setup assumption has converged to a standard ideal definition in the Universal Composability setting ($G_{\mathsf{att}}$, defined by Pass et al., Eurocrypt '17). However, it is unclear whether any real TEE design can actually implement this, or whether the diverse capabilities of today's TEE implementations will in fact converge to a single standard. Therefore, it is necessary for cryptographers and protocol designers to specify what assumptions are necessary for the TEE they are using to support the correctness and security of their protocol.
To this end, this paper 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 for comparing different classes of TEEs , particularly how a weaker TEE functionality can UC-emulate a stronger one given an appropriate mechanism to bridge the two. We introduce a new, "modular" definition of TEEsthat captures a broad range of pre-existing functionalities defined in the literature while maintaining their high level of abstraction. While our goal is not directly to model implementations of specific commercial TEE providers, our modular definition provides a way to capture more meaningful and realistic hardware capabilities. We provide a language to characterise TEE capabilities along the following terms:
- a set of trusted features available to the enclave;
- the set of allowed attacks for malicious interactions with the enclaves;
- the contents of attestation signatures.

We then define various possible ideal modular $G_{\mathsf{att}}$ functionality instantiations that capture existing variants in the literature, and provide generic constructions to implement stronger enclave functionalities from an existing setup. Finally, we conclude the paper with a simple example of how to protect against rollback attacks given access to a trusted storage feature.

## 1 Introduction

In recent years, programmable hardware-based Trusted Execution Environments (TEEs) have been made available by computer manufacturer for different market segments, including consumer and server CPUs. Their introduction has led them to be considered as a realistic component in the development of secure interactive protocols, for a range of diverse use cases. While actual real-world protocols deployments adopting TEEs have been limited, there has been a large number of academic publications exploring the feasibility of their use, both in the systems and cryptographic literature. Within the provable security community, a popular approach has been taking the existence of TEEs as a setup assumption. Hardware setup assumptions had previously considered simpler devices with limited computational power [49]. To capture the flexibility of a full fledged TEE, Pass, Shi, and Tramèr [76] (PST) formulate an ideal setup in the Generalised Universal Composability setting. Their functionality, the global attested execution functionality $G_{\mathsf{att}}$, captures the two core security claims of Trusted Execution: 1) programs run in an isolated environment (a secure "enclave") maintain confidentiality in the presence of an otherwise corrupted party; 2) the output of such a program is authenticated by "attestation". $G_{\mathsf{att}}$ models attestation as a (globally shared) signature over the program output and metadata, allowing any remote party to verify the value of attestation regardless of their access to an enclave.

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. As a consequence, the PST functionality does not expose a specific programming model, leading to publications with various (sometimes incompatible or unrealistic) assumptions about what features are available to the enclave program. It is also not clear whether the strong guarantees of the functionality can be met by any real implementation of TEEs, given the vast number of practical side-channels and physical attacks on the technology published since its release. It is thus necessary to question whether the promised guarantees can actually be delivered. This is an issue that most protocol designers that incorporate TEEs in their constructions conveniently choose to ignore by considering these 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. A salient example is given by Bhatotia et al. [13], who show how a weakened abstraction that allows malicious adversarial interference with an enclave's state could lead to loss of confidentiality, by mounting a *rollback attack*, in a protocol that would be secure in the $G_{\mathsf{att}}$-hybrid model. On the other hand, previous works [86, 41] 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 work 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. The goal of a feature oracle is to model the guarantees of the untrusted boundary between the trusted code running within the enclave and its access to the external (untrusted) world.
*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 values 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 choose the minimum set of features required by the enclave program, 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. 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 analyse 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 theoretical contribution. Constructing the next generation of practical TEE will require a much more system-focused approach than what is possible at this level of specificity.

The paper is structured as follows: in Section 2, we review some notions of Universal Composition and provide an overview of the existing TEE modelling from Pass, Shi, and Tramèr [76] and follow-up works. We then propose, starting from Section 3, a generic framework for specifying Trusted Execution functionalities with more granular interfaces. 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 values are included in the attestation measurement signed by each enclave. In Section 4 , 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 5 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, we sketch how combining a TEE setup with some attacks, and a protocol to defend against a portion of them, UC-emulate a setup with equivalent features, but without those attacks. We conclude by applying the latter compiler to an illustrative example in Section 6, where we sketch how to provide security in the presence of rollback attacks through a simple protocol that relies on access to trusted storage.

## 2 Background

### 2.1 Universal Composability

Universal Composability, introduced by Canetti [20], is a computational proof model that allows showing the security of protocols under concurrent composition.
UC is based on the computation model of Interactive Turing Machines (ITM) [48] 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 [68, 5, 58, 51, 18, 22] including different revisions to the original model (see [19, Appendix B]).

**Fundamentals** A succinct but comprehensive summary of the key components of UC can be found in [9, Section 2] and in [14, Appendix B]. In this section, we give a higher level overview of the model and discuss conventions adopted in the rest of the paper.
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 [19, 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 $\xi$-identity-bounded if the set of identities it can claim is restricted by $\xi$ (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 $\pi$ in the presence of adversary $\mathcal{A}$ (conversely $\mathsf{EXEC}_{\phi,\mathcal{S},\mathcal{Z}}$ is for the execution of the ideal functionality $\phi$ in the presence of simulator $\mathcal{S}$).

**Theorem 1 (UC emulation).** *For any PPT protocols $\pi, \phi$ and identity predicate $\xi$, we say that $\pi$ $\xi$-*UC-emulates* $\phi$ (or simply $\pi$ UC-emulates $\phi$ 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 $\xi$-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 $\pi$ that *realises* an ideal functionality $\mathcal{F}$, the security analysis of a new protocol $\rho$ that has $\pi$ as a subroutine can be carried out by replacing all of $\rho$'s call to subroutines running $\pi$ with calls to ideal functionality $\mathcal{F}$, which we denote as $\rho^{\pi \to \mathcal{F}}$. This new version of $\rho$ 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 $\rho$ that calls to a subroutine in $\pi$ or $\mathcal{F}$ satisfies $\xi$ and does not call instances of $\pi$ and $\mathcal{F}$ in the same session (we say that the protocol $\rho$ is $(\pi, \phi, \xi)$-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 (UC Composition Theorem).** *For any PPT protocols $\rho, \pi, \phi$ and predicate $\xi$, if $\rho$ is $(\pi, \phi, \xi)$-compliant, $\phi, \pi$ are both subroutine respecting and subroutine exposing, and $\pi$ $\xi$-UC-emulates $\phi$, then $\rho^{\pi \to \phi}$ UC-emulates $\rho$.*

Unfortunately, many interesting protocols, such as commitment schemes [23], secure two-party computation [24] or even authenticated channels [21], 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.

**Globality** While UC provides a powerful paradigm for reusable cryptographic proofs, composition imposes many restrictions over the base model as outlined in Theorem 2. To address the limitation of the UC theorem of subroutine-respecting interactions, Canetti and Rabin [25] 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. [29] formulate two new variants of Universal Composition, Extended UC and Generalised [3] 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 [26] 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 [9]. Camenisch, Drijvers, and Tackmann [17] also show that neither the UC or GUC composition theorems allow proving that a protocol $\rho^{\pi \to \mathcal{F}}$ can UC-emulate $\rho$ if $\pi$ is a subroutine of both $\rho$ and of another distinct ideal subroutine of $\rho$. They therefore propose a new recursive composition theorem for jointly subroutine respecting functionalities, multi-protocol UC.

Universal Composability with Global Subroutines [8] 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 $\pi$ with access to subroutine $\gamma$ 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 $\rho$ to access a single instance of $\pi$ and multiple of $\gamma$. $\mu$ is a shell only protocol that uses a directory ITI to redirect external communication from $\rho$ to the appropriate machines in $\pi$ or $\gamma$ (and conversely to the external machine that should receive a response). The following definition roughly corresponds to the EUC formulation of global functionalities:

**Definition 1.** *For protocols $\pi, \phi, \gamma$, we say that $\pi$ $\xi$-*UC-emulates $\phi$ in the presence of (global subroutine) $\gamma$ *if $M[\pi, \gamma]$ $\xi$-UC-emulates $M[\phi, \gamma]$*

As in the basic UC framework, the composition theorem follows, with some additional restrictions: $\pi$ and $\phi$ are allowed to break their subroutine- respecting behaviour to use the global subroutine $\gamma$ (we say they are $\gamma$-subroutine respecting), and $\gamma$ itself does not depend on $\phi$ as one of its subroutines (we say that $\gamma$ is $\phi$-regular). These requirements allow the use of the shared state subroutine without provoking circular dependencies that would prevent a clean cut replacement [4].

**Theorem 3 (Universal Composition with Global Subroutines).** *For any subroutine-exposing protocols $\rho, \phi, \pi, \gamma$ where*
- *$\gamma$ is subroutine respecting and $\phi$-regular,*
- *$\pi, \phi$ are $\gamma$-subroutine respecting,*
- *$\rho$ is $(\pi, \phi, \xi)$-compliant, $(\pi, M[\phi, \gamma], \xi)$-compliant and $(\pi, M[\pi, \gamma], \xi)$-compliant;*

*if $\pi$ $\xi$-UC-emulates $\phi$ in the presence of $\gamma$ (per Definition 1), then $\rho^{\phi \to \pi}$ UC-emulates $\rho$.*

The above theorems can be used to recover EUC statements in the literature by formulating an appropriate identity bound. While most of the existing work focus on ideal functionalities as global subroutine, Badertscher, Hesse, and Zikas [7] show that UCGS does not universally preserve the composition theorem from [26] 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 $\gamma$ that realises it), the simulator $S$ in the emulation of a $G$-hybrid functionality $F$ by some

[3] commonly misattributed as Global UC

[4] This type of recursive composition is implemented in multi-protocol UC [17]; however the composition theorem of that work is not compatible with Theorem 2

protocol $\pi$ might no longer be possible in the $\gamma$-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 $\gamma$ UC-emulates ideal functionality $G$, and $G$ UC-emulates $\gamma$) 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 $\gamma$ (S is an *admissible* simulator).
Canetti et al. [28] later observes that the replacement statement also holds if protocol $\gamma$ replaces the protocol that combines $G$ with the simulator from the $\gamma$ UC-emulates $G$ experiment, and thus any $F$ using that combined protocol as a global subroutine can be replaced with $\gamma$.
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 [8, Definition 4]. Additional remarks about our notation when we present UC protocols follows.

**Notation** We now list additional convention taken by our pseudocode for the remainder of this 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 above; any such usage is self-contained to this section, but we refer the reader to [19, 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** ($\textsc{msg}, args$), 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 $m = (\textsc{msg}, args)$. 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[5]. 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.
Our formulations in this work rely on structured protocols, as defined in [19, 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 *command*(*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 $\mathsf{prog}$ (including command and arguments) without executing them (i.e. by advancing only the head of the shell over the tape). Conversely, when an ITI $\rho$ in a structured protocol (see 2.1) contains pseudocode

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

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

## 2.2 Trusted Execution Environments

Trusted Execution Environments have generally been the domain of the system security research community. Costan and Devadas [37] first attempted to bridge the knowledge gap to allow cryptographers to understand the internals and guarantees of Intel SGX, the first widely available and commercially successful TEE. We assume the reader is familiar with the high level guarantees of a TEE, and refer to that work for an in-depth explanation of the internals of one of its most popular instantiation.
Since then, a variety of works have attempted to formalise TEEs for the purposes of cryptographic protocol design.

- Sinha et al. [79] and Sun and Lei [82] 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. [94] propose a model in the Tamarin prover to automate security proofs of protocols that rely on TEEs.
- Sinha et al. [80] 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. [40] reduce leakage resilience their TEE-enabled application to the safety of the type-system of the Rust programming language

[5] 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

- Antonino, Woloszyn, and Roscoe [4] 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 [89] 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. [81] 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. [59] later extend the TAP model to capture memory sharing capabilities between enclaves, and Gaddamadugu [47] provides a machine checked-proof using this variant.
- Fisch et al. [43] define a game-based model of SGX to prove the security of their protocol
- Barbosa et al. [11] 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 [52] provides a SAPIC/Tamarin machine-checked simplification of this model, and Bahmani et al. [10] later realise a secure MPC protocol using the model in a simulation-based proof.
- Lu et al. [63] provides a simulation based security definition for a TEE with a partial corruption model.
- Pass, Shi, and Tramèr [76] gives a definition of TEEs under the Universal Composability setting.

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.

**TEEs under Universal Composability** While various works exist to model HSM-like functionality in UC (e.g. see [55]), and some initial work has been proposed by Canetti et al. [30] to give a UC treatment of validating the security guarantees of generic hardware constructions (including protecting against side-channel attacks), Pass, Shi, and Tramèr [76] provide the first UC formulation of TEEs. Their $G_{\mathsf{att}}$ functionality (fully reproduced below in Figure 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. $G_{\mathsf{att}}$ 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, $G_{\mathsf{att}}$ 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 1 diverges from the original one in that we let $\mathsf{vk}$ be a global variable, accessible by enclave programs as $G_{\mathsf{att}}.\mathsf{vk}$. This allows us to use $G_{\mathsf{att}}$ 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 $\mathsf{reg}$); 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 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.

$G_{\mathsf{att}}$ is a Global Functionality in GUC [29] where the only meaningful global state shared between all protocols is the attestation verification key. This is a simplification over the EPID

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

| State variables | Description |
|---|---|
| $\mathsf{vk}$ | Master verification key, available to enclave programs |
| $\mathsf{msk}$ | 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 $(\mathsf{output}, \sigma)$
```

**Fig. 1.** The $G_{\mathsf{att}}$ functionality of [76]

attestation protocol used in the original version of SGX [77], 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 [76] 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 $G_{\mathsf{att}}$ in the (G)UC framework [96, 34, 32, 95, 33, 92, 62, 60, 36, 50, 65, 61, 91, 53, 56, 45, 46, 73, 12, 35] or using [64] the Abstract Cryptography framework [69], and it as provides a basis for formalising TEEs in property-based definitions [70, 66, 42, 93, 44, 38, 83].
Additionally, some attempts have been made to relax the $G_{\mathsf{att}}$ functionality for the purposes of capturing TEE vulnerabilities. Tramer et al. [84] introduced the concept of transparent enclaves to model confidentiality leaks in an enclave program (formalised under GUC in [75, Section 8.1] ). The transparent enclave functionality behaves exactly as $G_{\mathsf{att}}$, 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 [41] 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 cryptographic 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 [63], 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 $\mathsf{reg}$).
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.
Bhatotia et al. [13] provides a further weakened UCGS version of $G_{\mathsf{att}}$ that allows an adversary to conduct rollback and forking attacks. Their functionality keeps track of enclave states in a tree data structure and allows a corrupted party to select an arbitrary node in the tree to load the state from as part of a RESUME operation. This new weaker setup can be shown to no longer be sufficient to guarantee the security of a protocol that includes stateful enclaves.

## 3 A modular $G_{\mathsf{att}}$ Setup

As an ideal functionality, the $G_{\mathsf{att}}$ formalisation described in the previous section does not really explain how enclave execution takes place in any detail. This formulation of TEEs does not explicitly expose any 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 $G_{\mathsf{att}}$, Pass, Shi, and Tramer [75, 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.
This 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. This high level model does however conceal how realistic hardware component might fail, preventing the protocol designer to take such a scenario into account. A more careful approach would then consider the functionality provided by $G_{\mathsf{att}}$ 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 $G_{\mathsf{att}}$ could be considered a "standard" enclave interface, it is possible to imagine additional API calls available to the enclaves, for example a trusted clock [31], monotonic counters[31, 67], secure access to GPU compute resources [85, 88, 97] 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 $G_{\mathsf{att}}$ functionality from [76] (henceforth referred to as $G_{\mathsf{att}}^{PST}$) 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 instantiation of a TEE $G_{\mathsf{att}}^{mod}$, 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 $G_{\mathsf{att}}^{mod}$, 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 parameterise their chosen version of $G_{\mathsf{att}}^{mod}$. 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 $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ (Fig. 2) and the original $G_{\mathsf{att}}^{PST}$ functionality, introduced in Section 2.2
We iterate on the work of [13] to more carefully follow the conventions and formality of modern UC versions compared to $G_{\mathsf{att}}^{PST}$. In particular, we now model enclaves as structured ITI subroutines to the $G_{\mathsf{att}}^{mod}$ functionality. On installation of an enclave, the functionality spawns a new ITI subroutine with composite extended identity[6] $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$, encoding the program $\mathsf{prog}$, oracles $\mathbb{O}, \mathbb{A}$, the unique enclave ID $\mathsf{eid}$, the identity $\mathsf{pid}$ for the party that installed the enclave, and the claimed session identity $\mathsf{idx}$. 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 $G_{\mathsf{att}}^{mod}$ is known as a *shell*, and a second subroutine with code $\mathsf{prog}$ 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 3 for a graphical representation). Running enclaves as separate subroutine 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

[6] 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

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

| State variables | Description |
|---|---|
| $\mathsf{vk} \leftarrow \epsilon$ | Master verification key |
| $\mathsf{Sign} \leftarrow \epsilon$ | 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 $\mathsf{vk}$
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 $\mathsf{pid}$ 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 $\mathsf{MissingInstructionError}$
  generate nonce $\mathsf{eid} \xleftarrow{\$} \{0,1\}^\lambda$, store $\mathcal{T}[\mathsf{eid}, \mathsf{pid}] = (\mathsf{idx}, \mathsf{prog})$
  send INSTALL to $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
  return $\mathsf{eid}$
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 $\mathsf{attack} = \epsilon \lor \mathsf{pid}$ is not corrupted then
      send $\mathsf{input}$ to $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ and receive $\mathsf{output}$
  else
      assert $\mathsf{attack} \in \mathbb{A}$
      send $(\mathsf{attack}, \mathsf{input})$ to $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ and receive $\mathsf{output}, \mathsf{aux}$
      if $\mathsf{aux} \neq \epsilon$ then
          query $\mathcal{A}$ with $(\mathsf{attack}, \mathsf{aux})$ and receive the reply CONTINUE
  let $\mathsf{meas} \leftarrow \mathbb{S}(\text{configuration of } \mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}])$
  let $\sigma \leftarrow \mathsf{Sign}(\mathsf{meas}), \mathcal{S}[\sigma] \leftarrow \mathcal{S}[\sigma] \ || \ \mathsf{meas}$
  return $(\mathsf{output}, \sigma)$
```

**Fig. 2.** Global functionality $G^{mod}_{\mathsf{att}}$

in that the shell rejects any input message not sent through the $G_{\mathsf{att}}^{mod}$ 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, $G_{\mathsf{att}}^{mod}$ needs to be activated with sufficient import to activate those subroutines).

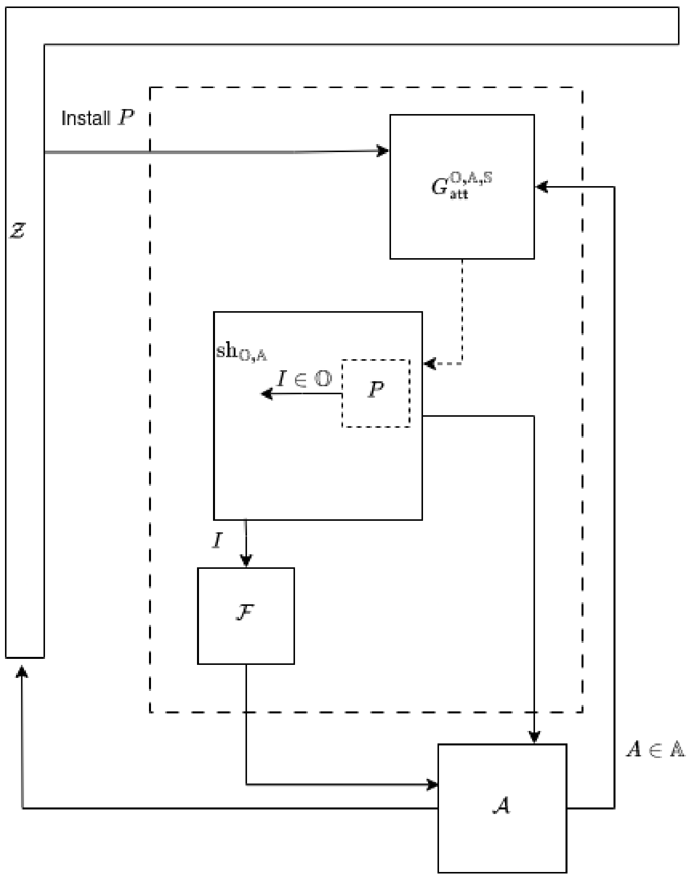

**Fig. 3.** When a program with code $P$ is installed on a $G_{\mathsf{att}}^{mod}$ enclave, the functionality spawns a new structured protocol subroutine with shell $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[]$ and body $P$. For some interfaces $I \in \mathbb{O}$, the shell will outsource its computation to some external functionality $\mathcal{F}$. The adversary $\mathcal{A}$ can interact with the enclave shell for any attacks $A \in \mathbb{A}$ through $G_{\mathsf{att}}^{mod}$. Both $\mathrm{sh}_{\mathbb{O},\mathbb{A}}[]$ and $\mathcal{F}$ can leak additional information to $\mathcal{A}$

We parameterise each instance of $G_{\mathsf{att}}^{mod}$ by the static sets $\mathbb{O}, \mathbb{A}$ which capture feature and adversarial oracles respectively. On installation of a new enclave, $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$. 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 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 $G_{\mathsf{att}}^{PST}$ 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 [75, Section 8.4]). We further relax the attestation mechanism of the $G_{\mathsf{att}}^{PST}$ 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) $G_{\mathsf{att}}^{mod}$ 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, $G_{\mathsf{att}}^{mod}$ implements signature verification by maintaining a map $\mathcal{S}$ of all signed strings and corresponding signatures generated by $\mathsf{Sign}$. 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 $G_{\mathsf{att}}^{PST}$, 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 $\Sigma$, and initialises a $G_{\mathsf{att}}^{mod}$ instance with closure $s(\mathsf{meas}) = \Sigma.Sign(sk, \mathsf{meas})$, such that on a RESUME call, $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ functionality is running code of type $\mathsf{prog} = (\mathsf{app}, \mathsf{runtime})$ for a specific $\mathsf{runtime}$ 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 $G_{\mathsf{att}}^{mod}$ only installing and executing $\mathsf{app}$ as part of its enclave. Of course, $\mathsf{app}$ 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 $G_{\mathsf{att}}^{PST}$ UC-emulates $G_{\mathsf{att}}^{mod}$ for the sets of oracles and measurement function that correspond to $G_{\mathsf{att}}^{PST}$ (which we describe in the next section). We construct a simulator that selects the exact signature scheme specified in $G_{\mathsf{att}}^{PST}$. Note that the opposite direction $G_{\mathsf{att}}^{mod}$ UC-emulates $G_{\mathsf{att}}^{PST}$ 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 $G_{\mathsf{att}}^{mod}$, 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 $G_{\mathsf{att}}^{PST}$. 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. [28] 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 $\rho$, to replace any subroutine call from $\rho$ to $G$ with a call to the combined subroutine of $G'$ and $S$. This enables us to port any existing proofs that rely on $G_{\mathsf{att}}^{PST}$ (provided that the proof is valid under UCGS rather than GUC) into our new model, by simply replacing $G_{\mathsf{att}}^{PST}$ with the combination of the $G_{\mathsf{att}}^{mod}$ instance with equivalent $\mathbb{O}, \mathbb{A}$ oracles (which we describe in Section 4.1), and the simulator that at instantiation chooses the precise $G_{\mathsf{att}}^{PST}$ signature scheme over the usual $(\mathsf{idx}, \mathsf{pid}, \mathsf{prog}, \mathsf{output})$ measurement produced by $\mathbb{S}$.

## 4 Defining a $G_{\mathsf{att}}^{mod}$ Zoo

We now provide the definition for several sets of $G_{\mathsf{att}}^{mod}$ oracle instantiations, aiming to capture all existing variants of $G_{\mathsf{att}}^{PST}$ 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 $G_{\mathsf{att}}^{mod}$ 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 4.4), 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 $G_{\mathsf{att}}^{mod}$ that use the same attestation signature function $\mathbb{S}$ as $G_{\mathsf{att}}^{PST}$ i.e. anonymous attestastions over $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output})$, unless stated otherwise .

### 4.1 $G_{\mathsf{att}}^{PST}$

We begin by reformulating $G_{\mathsf{att}}^{PST}$ in the language of $G_{\mathsf{att}}^{mod}$. While this is not made explicitly in the original work, $G_{\mathsf{att}}^{PST}$ 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 registered 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 $G_{\mathsf{att}}^{PST}$ paper[75, page 23], but we include it because many $G_{\mathsf{att}}^{PST}$-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. $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{PST}$ 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 [75, 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 $G_{\mathsf{att}}^{PST}$ 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 $G_{\mathsf{att}}^{mod}$ functionality, so no additional attack is required.
For capturing $G_{\mathsf{att}}^{PST}$ under $G_{\mathsf{att}}^{mod}$, 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 $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$, where the PID is a concatenation of the enclave identifier generated by $G_{\mathsf{att}}^{mod}$ 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 section.

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

The shell is defined for $\mathbb{O} = \mathbb{O}^{\text{std}} \cup \{\text{AttestVerify}\}$ and $\mathbb{A} = \{\}$
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
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 $\mathsf{i} = \text{AttestVerify}(\sigma, m)$ then
          send $(\text{VERIFY}, \sigma, m)$ to $G_{\mathsf{att}}^{mod}$ and receive $v$
          append $v$ to subroutine output tape for virtual ITI
      else if $\mathsf{i} = (\textbf{return } v)$ then
          return $v$ with source $(\text{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
```

The shell receives message INSTALL when it is first created from $G_{\mathsf{att}}^{mod}$, 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 section, 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 $\mathsf{input}$ the shell receives from $G_{\mathsf{att}}^{mod}$ must be the argument of a RESUME call, since the adversary is not able to give an attack message. Rather than writing $\mathsf{input}$ 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 $\mathsf{i}$ that would be executed if it was activated with $\mathsf{input}$ (we denote this as “begin executing $\mathsf{input}$ on $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$”). With this the shell enters its main loop: depending on what type of instruction $\mathsf{i}$ is, it executes $\mathsf{i}$ 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 $\mathbb{O}^{\text{std}}$. In this case, the shell activates $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ with input $\mathsf{i}$; 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 $\mathsf{i}$ from the updated configuration.
When the instruction is of type $\text{AttestVerify}(\cdot)$, the shell does not activate $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$, but rather sends a message to $G_{\mathsf{att}}^{mod}$ 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 $\mathsf{i}$ for the enclave is to return some value, the shell forwards it to $G_{\mathsf{att}}^{mod}$, overwriting the sender-id of the outgoing message with its own extended identity. The shell thus yields activation back to $G_{\mathsf{att}}^{mod}$, which proceeds with generating the attestation by calling $\mathbb{S}$ on the configuration of $(\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx})$.

## 4.2 Accessing a Clock

A natural extension of $G_{\mathsf{att}}^{PST}$, which the original paper uses to realise fair MPC [75, 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 $G_{\mathsf{att}}^{mod}$ functionality that supports feature oracles $\mathbb{O} = \mathbb{O}^{\text{std}} \cup \{\text{ReadRound}, \text{IncRound}\}$ 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.

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

The shell is defined for $\mathbb{O} = \mathbb{O}^{\text{std}} \cup \{\text{ReadRound}, \text{IncRound}\}$ and $\mathbb{A} = \{\}$
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
  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 $\mathsf{i} \in \mathbb{O}^{\text{std}}$ then
        allow $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$ to execute $i$
     else if $\mathsf{i} =$ 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 $\mathsf{i} =$ 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 $\bot$
    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 Interrupting computation

As a first attempt to show how to capture an attack oracle, we now model a new version of $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ 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:

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

The shell is defined for $\mathbb{O} = \mathbb{O}^{\mathrm{std}} \cup \{\text{Store}, \text{Fetch}\}$ and $\mathbb{A} = \{\text{Abort}, \text{Continue}\}$
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 |
|---|---|
| $mem \leftarrow \epsilon$ | 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 $\mathsf{i} \in \mathbb{O}^{\mathrm{std}}$ 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 $i = \text{Fetch}$ 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 $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message Abort from $G_{\mathsf{att}}^{mod}$:
  if $\mathsf{halt} = \bot$ 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 $\mathsf{halt} = \bot$ 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 $\mathsf{eid}$ run by party $\mathsf{pid}$ in session $\mathsf{idx}$ 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 $(\textsc{resume}, \mathsf{eid}, \epsilon, a \in \mathbb{A})$ from corrupted party $\mathsf{pid}$ to $G_{\mathsf{att}}^{mod}$. 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 $\mathsf{prog}$, the enclave can be resumed later on, and recover some partial state from the last value successfully stored to $\mathsf{mem}$, if any. The $\mathcal{A}$ can call the attack oracles at any other point, without the enclave trying to access memory (i.e. when $\mathsf{halt} = \bot$); 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 $\mathsf{halt}$ 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.

## 4.4 Rollback Attacks

While the previous version of $G_{\mathsf{att}}^{mod}$ 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.
Bhatotia et al. [13] introduces a new variant of $G_{\mathsf{att}}$ that allows state continuity attacks, $G_{\mathsf{att}}^{\mathsf{rollback}}$. The functionality 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.
Since $G_{\mathsf{att}}^{mod}$ no longer tracks the state of an enclave in a table $\mathcal{T}$, an instance of $G_{\mathsf{att}}^{mod}$ 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 $\textsc{resume}$ 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 $\mathsf{pid}$ is corrupted then send $(\mathrm{ITER}, \emptyset, c)$ 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 $\mathsf{pid}$ is corrupted then send $(\mathrm{ITER}, c, c')$ to $\mathcal{A}$
    $c \leftarrow c'$
On message $(\mathrm{ROLLBACK}, (i, \mathsf{input}))$ from $G_{\mathsf{att}}^{mod}$:
    execute $(\mathsf{out}, (\mathrm{FORK}, c, i, i')) \leftarrow (\mathrm{FORK}, i, \mathsf{input})$
    $c \leftarrow i'$
    return $(\mathsf{out}, (\mathrm{ROLLBACK}, i, i'))$
On message $(\mathrm{FORK}, (i, \mathsf{input}))$ from $G_{\mathsf{att}}^{mod}$:
    if virtual ITI $(\mathsf{prog}, (\mathsf{eid}||i, \mathsf{idx}))$ exists then
        $\mathsf{out} \leftarrow \epsilon$
        if $\mathsf{input} \neq \epsilon$ 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 $(\mathsf{out}, (\mathrm{FORK}, c, i, i'))$
```

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 (RESUME, $\mathsf{eid}, \mathsf{input}$, Rollback) from corrupted party $\mathsf{pid}$ to $G_{\mathsf{att}}^{mod}$), $c$ 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 $G_{\mathsf{att}}^{mod}$ 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, $G_{\mathsf{att}}^{mod}$ waits for the adversary to issue a CONTINUE message to finalise the return value and produce attestation (otherwise the RESUME call for $G_{\mathsf{att}}^{mod}$ 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.

## 4.5 Modelling side channels

As we have discussed in the Background section 2.2, some previous works in the literature have extended the $G_{\mathsf{att}}^{PST}$ model to capture additional types of side-channel attacks. We now adapt those extensions into $G_{\mathsf{att}}^{mod}$ shells.

*Transparent enclaves* Tramer et al. [84] provides a (local) UC functionality for attested execution with no confidentiality guarantees, later extended in [75, 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 $G_{\mathsf{att}}^{PST}$: 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 $G_{\mathsf{att}}^{mod}$, 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 [84] and [75, 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 issues the TransparentLeak attack, all further values of the tape are leaked.

```
sh_{O,A}[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
  $\mathsf{transparent} \leftarrow \perp$
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 (LEAK, 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 (TRANSPARENTLEAK, input) from $G_{\mathsf{att}}^{mod}$:
  set $\mathsf{transparent} \leftarrow \top$
  if $\mathsf{input} \neq \epsilon$ then
      parse $(cmd, args) \leftarrow \mathsf{input}$, return $cmd(args)$
  else
      return ($\epsilon$, 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 [41] introduce two relaxations over the $G_{\mathsf{att}}$ functionality that aim to capture a middle ground between the side-channel free $G_{\mathsf{att}}^{PST}$ 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 [57].

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

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 $(\mathcal{F}_{\mathsf{crypto}}, (\mathsf{idx}, \bot))$ 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 $\mathsf{i}$ 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 $\mathsf{i} = (\mathrm{KeyExchange}, \mathsf{pid}', \mathsf{eid}')$ then
      set $\mathsf{halt} \leftarrow \top$
      send GenExp to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{ExpoPointer}, ptr^e, g^e)$
      if $\mathsf{pid}$ is corrupted then
        query $\mathcal{A}$ with $(\mathrm{KeyExTo}, \mathsf{pid}', \mathsf{eid}')$ and receive the reply continue
      if $\mathcal{E}[\mathsf{pid}', \mathsf{eid}'] = \bot$ then
        // no stored keyshare for $\mathsf{eid}'$, we are the initiatior
        send $(\mathrm{KeyEx}, g^e)$ to $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}'||\mathsf{pid}', \text{“att”}||\mathsf{idx}))$ and await
        while next message on the input tape is not $(\mathrm{KeyEx}, \mathsf{pid}', \mathsf{eid}', h)$ do ignore
        send $(\mathrm{BlockGroupElement}, h)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive OK
      else
        // $\mathsf{eid}'$ was the key exchange initiatior, we already have $h$
        $h \leftarrow \mathcal{E}[\mathsf{pid}', \mathsf{eid}']$
      send $(\mathrm{GenDHKey}, ptr^e, h)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Pointer}, ptr^{dhk})$
      send $(\mathrm{Derive}, ptr^{dhk}, \texttt{unauth-key})$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Pointer}, ptr^{sk})$
      set $\mathsf{halt} \leftarrow \bot$
      append $ptr^{sk}$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = \mathrm{SKEGen}$ then
      send $(\mathrm{New}, \texttt{unauth-key})$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Pointer}, ptr)$
      append $ptr$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\mathrm{SKEEnc}, ptr, m)$ then
      send $(\mathrm{Enc}, ptr, m)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Ciphertext}, hdl)$
      append $hdl$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\mathrm{SKEDec}, hdl, ct)$ then
      send $(\mathrm{Dec}, ptr, ct)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Plaintext}, m)$
      append $m$ to subroutine output tape for virtual ITI $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    else if $\mathsf{i} = (\mathrm{ReleaseKey}, ptr)$ then
      send $(\mathrm{Retrieve}, ptr)$ to $\mathcal{F}_{\mathsf{crypto}}$ and receive $(\mathrm{Key}, k)$
      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 $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
On message Halt from $G_{\mathsf{att}}^{mod}$:
  set $\mathsf{halt} \leftarrow \top$
  return
On message $(\mathrm{KeyEx}, h)$ from $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}'||\mathsf{pid}', \text{“att”}||\mathsf{idx}))$:
  if $\mathsf{halt} = \bot$ then
    // we are not waiting for key exchange to complete;
    // eid' is the initiator
    send (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 [41] define an enclave with access to both $\{\text{KeyExchange}, \text{SKEGen}, \text{SKEEnc}, \text{SKEDec}, \text{ReleaseKey}\} \in \mathbb{O}$ and $\text{TransparentLeak} \in \mathbb{A}$ 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 $\mathbb{A} = \{\text{CompleteLeak}\}$
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
  rec ← []
On message (RESUME, input) from $G_{\mathsf{att}}^{mod}$:
  begin executing input on (prog, (eid, idx))
  for next instruction i on virtual ITI do
    if i ∈ $\mathbb{O}^{\text{std}}$ then
      allow (prog, (eid, idx)) to execute $i$
    else if i =(return $v$) then
      rec ← rec || (input, $args$, virtual ITI work tape)
      return $v$ with source ($\text{sh}_{\mathbb{O},\mathbb{A}}$[prog], (eid||pid, "att"||idx))
On message CompleteLeak from $\mathcal{A}$:
  return 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.

## 4.6 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 $\text{Censor}_\text{Q}$ is parameterised by $Q$.

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

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

```
On message JOIN from $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$:
  $P \leftarrow P \cup (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
  send $(\textsc{Join}, (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$ to $\mathcal{A}$
On message $(\mathrm{cmd}, v)$ from $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$:
  if eid is running on a corrupted party then
    query $\mathcal{A}$ with $(\textsc{Read}, (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$ and receive the reply Block, $b$
  if $b \neq \top \wedge \frac{|C|}{|P|} < Q$ then
    if cmd=WRITE then
      if $w = \bot$ then $w \leftarrow P$
      if $w \neq P \vee P \in C$ then return Fail
      $V \leftarrow V \parallel v$
    send $(\textsc{cmd}, V, (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$ to $\mathcal{A}$ return $V$
  elsereturn Fail
On message HEALTHCHECK from $(\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$:
  return $|P|, |C|$
On message $(\textsc{Censor}, (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx})))$ from $G_{\mathsf{att}}^{mod}$:
  if eid is running on a corrupted enclave then
    $C \leftarrow C \parallel (\mathsf{prog}, (\mathsf{eid}, \mathsf{idx}))$
    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[7] 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.

## 5 Relationships between $G_{\mathsf{att}}^{mod}$ variants

Having defined a variety of different $G_{\mathsf{att}}^{mod}$ 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:

[7] 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.

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 $G_{\mathsf{att}}^{mod}$ which sign over the same measurement functions, a "weaker" setup ($G_{\mathsf{att}}$) that has either fewer features or more attacks can UC-emulate the stronger one ($G'_{\mathsf{att}}$). If there is a "wrapper" protocol around $G_{\mathsf{att}}$ 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 $G_{\mathsf{att}}$ setup is at least as strong as $G'_{\mathsf{att}}$.
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.

## 5.1 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 $G_{\mathsf{att}}$ and $G'_{\mathsf{att}}$ (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 $G_{\mathsf{att}}$ and $G'_{\mathsf{att}}$. We now define a new "wrapper" protocol $\mathcal{W}$ which uses $G_{\mathsf{att}}$ as a subroutine and UC-realises $G'_{\mathsf{att}}$ by implementing the interface for $I$ in the real world.
$\mathcal{W}$ takes the same parameters as $G_{\mathsf{att}}^{mod}$, 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 $G_{\mathsf{att}}^{mod}$-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 $\mathrm{W}^{I}[\cdot]$.
$\mathrm{W}^{I}[\mathsf{prog}, \mathrm{nextmsg}]$ is a "wrapper" enclave that instruments $\mathsf{prog}$ with additional code such that, when $\mathsf{prog}$ 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 $\mathsf{prog}$ 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 [16]). 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 $\mathsf{Steel}$ protocol of [13].
The $\mathcal{W}$ protocol (Figure 4) proceeds as follows. During initialisation, it calls the $\mathsf{map}^{\mathsf{R}}$ function to produce a list of supporting enclaves $\mathrm{E}_i^R$ run by a subset of $\mathsf{reg}$ parties, initialises $G_{\mathsf{att}}^{mod}$ with the appropriate parameters, and requests each selected party to install the wrapped $\mathrm{E}_i^R$. It then returns a public list of all assisting parties and the associated enclave IDs.
On a call from $\mathsf{pid}_i$ to install some program $\mathsf{prog}_{\mathsf{i}}$, if $\mathsf{prog}_{\mathsf{i}}$ does not include any call to $I$, it installs a wrapped version with dummy next message function $\epsilon$. Otherwise, it runs $\mathsf{map}^{\mathsf{L}}()$

```
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 $P$:
  $[(\mathrm{E}_1^R, \mathsf{pid}_1), \dots, (\mathrm{E}_n^R, \mathsf{pid}_m)] \leftarrow \mathsf{map}^{\mathsf{R}}(\mathsf{reg})$
  send INITIALISE to $G_{\mathsf{att}}^{mod}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}]$
  let $\bar{\mathrm{E}}^R \leftarrow []$
  for $i \in \{0, \dots, n\}$ do
    send $(\textsc{installRequest}, \mathrm{E}_i^R)$ to $\mathsf{pid}_i$ and receive $\mathsf{eid_i}$
    $\bar{\mathrm{E}}^R \leftarrow \bar{\mathrm{E}}^R \parallel \mathsf{pid}_i, \mathsf{eid_i}$
  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 $(\textsc{verify}, \sigma, m)$ 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 $(\textsc{verify}, \sigma, m)$ to $G_{\mathsf{att}}^{mod}$ and receive $v$ and return v
On message $(\textsc{install}, \mathsf{prog})$ from a party $P$ where $P.\mathsf{pid} \in \mathsf{reg}$:
  if $I \in \mathsf{prog}$ then
    $(\mathrm{nextmsg}_{\emptyset}, (\mathrm{E}_1^L, \dots, \mathrm{E}_n^L)) \leftarrow \mathsf{map}^{\mathsf{L}}(\bar{\mathrm{E}}^R)$
    let $\bar{\mathrm{E}}^L \leftarrow []$
    for $i \in \{1, \dots, n\}$ do
      send $(\textsc{install}, \mathrm{E}_i^L)$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{eid_i}$
      send $(\textsc{resume}, \mathsf{eid_i}, \textsc{init})$ to $G_{\mathsf{att}}$
      $\bar{\mathrm{E}}^L \leftarrow \bar{\mathrm{E}}^L \parallel \mathsf{eid_i}$
    let $\mathrm{nextmsg}(x) \leftarrow \mathrm{nextmsg}_{\emptyset}(x, \bar{\mathrm{E}}^L)$
    send $(\textsc{install}, \mathrm{W}^I[\mathsf{prog}, \mathrm{nextmsg}])$ to $G_{\mathsf{att}}$ and receive $\mathsf{eid_{prog}}$
    send $(\textsc{resume}, \mathsf{eid_{prog}}, \textsc{init})$ to $G_{\mathsf{att}}$
  else
    send $(\textsc{install}, \mathsf{sid}, \mathrm{W}^I[\mathsf{prog}, \epsilon])$ to $G_{\mathsf{att}}$ and receive $\mathsf{eid_{prog}}$
  return $\mathsf{eid_{prog}}$
On message $(\textsc{resume}, \mathsf{eid}, \mathsf{input})$ from a party $P$ with $\mathsf{pid}_i$:
  send $(\textsc{resume}, \mathsf{eid}, \mathsf{input})$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{out}, \sigma$
  while $\mathsf{out} = (\textsc{resumeRequest}, \mathsf{pid}, \mathsf{eid}', v)$ do
    if $\mathsf{pid} = \mathsf{pid}_i$ then
      $(\mathsf{out}, \sigma') \leftarrow \textsc{resume}(\mathsf{eid}', (v, \sigma, \top)))$
    else
      send $(\textsc{resumeRequest}, \mathsf{eid}', (v, \sigma))$ 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 $(\textsc{installRequest}, \mathsf{prog})$ from a party $P \in \mathsf{reg}$:
  send $(\textsc{install}, \mathsf{prog})$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{eid}$
  return $\mathsf{eid}$
On message $(\textsc{resumeRequest}, \mathsf{eid}, \mathsf{input})$ from a party $P \in \mathsf{reg}$:
  send $(\textsc{resume}, \mathsf{eid}, \mathsf{input}))$ to $G_{\mathsf{att}}^{mod}$ and receive $\mathsf{output}, \sigma$
  return $(\mathsf{output}, \sigma)$
```

**Fig. 4.** The wrapper protocol

to produce a list of local assisting enclave programs to be installed by the same party, and a next message function $\text{nextmsg}_\emptyset$. The party installs all such enclaves, runs their initialisation subroutine, and creates a new message function nextmsg that is a wrapper around $\text{nextmsg}_\emptyset$ aware of the assisting parties enclave IDs. $\mathsf{map}^\mathsf{L}$ makes nextmsg and all $\mathrm{E}_i^L$ programs aware of the enclave IDs for any $\mathrm{E}^R$ 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 $\mathsf{pid}_i$ to execute command cmd on arguments $args$ for enclave $\mathrm{W}^I[\mathsf{prog_i}, \text{nextmsg}]$, the enclave wrapper (described in Figure 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 $\mathsf{pid}_i$, activates one of its local enclaves, including $\mathrm{W}^I[\mathsf{prog_i}, \text{nextmsg}]$ 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. $\mathsf{pid}_i$ 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 $G_\mathsf{att}^{mod}$ 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 $G'_\mathsf{att}$ (see Figure 6 for a graphical representation).
We now provide the following conjecture:

*Conjecture 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 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 $G_\mathsf{att}$ as a subroutine, UC-emulates $G'_\mathsf{att}$

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 $G_\mathsf{att}$ UC-emulates $G'_\mathsf{att}$. 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 Pass, Shi, and Tramèr [76] and Bhatotia et al. [13]).
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 wrapper $\mathrm{W}^I[\cdot]$ can implement $I$ without relying on any external assistance. During the global functionality initialisation phase, the simulator observes the signature algorithm $s$ chosen by the environment through the dummy adversary, and provides $G'_\mathsf{att}$ 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 $\mathsf{prog}$, and replaces it with an identifier for $\mathrm{W}^I[\mathsf{prog}, \text{nextmsg}]$ (for an appropriate value of nextmsg), as discussed in Section 3. Attestations produced by

```
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 $\mathsf{input}$ 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 $\mathsf{i}$ is delegated to the higher order shell
              allow $(\mathsf{prog}, (\mathsf{eid}||\text{“wrapped”}, \mathsf{idx}))$ to execute $\mathsf{i}$
          else
              $(\mathsf{pid_j}, \mathsf{eid_j}, v) \leftarrow \mathrm{nextmsg}(\text{tapes of virtual ITI})$
              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 $\mathrm{nextmsg}()$ returns $(\perp, \perp, v)$
  else
      // $r = \top$ as the result of an $I$ computation, execute code in subroutine $\mathrm{cmd}$
      execute $\mathrm{cmd}(args)$
      return $\mathrm{nextmsg}(\text{tapes of virtual ITI})$
```

**Fig. 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

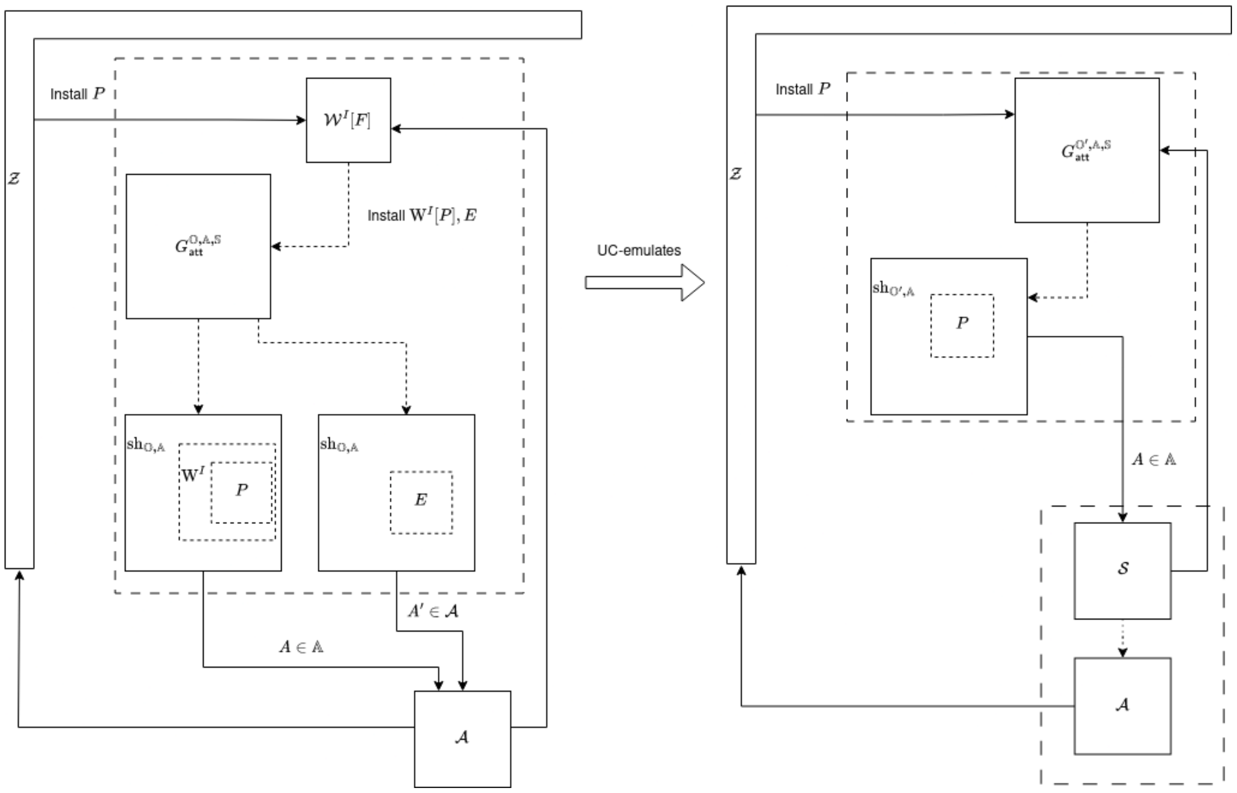
Install $P$
$\mathcal{W}^I[F]$
$\mathcal{Z}$
$G_{\mathsf{att}}^{\mathbb{O},\mathbb{A},\mathbb{S}}$
Install $\mathrm{W}^I[P], E$
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}$
$\mathrm{W}^I$
$P$
$\mathrm{sh}_{\mathbb{O},\mathbb{A}}$
$E$
$A' \in \mathcal{A}$
$A \in \mathbb{A}$
$\mathcal{A}$
UC-emulates
Install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

**Fig. 6.** Protocol $\mathcal{W}$ can add a shell to $G_{\mathsf{att}}^{\mathbb{O},\mathbb{A}\mathbb{S}}$ enclaves to UC-emulate the missing feature oracles $I$ from $G_{\mathsf{att}}^{\mathbb{O}',\mathbb{A}\mathbb{S}}$

an enclave prog in the $G'_{\mathsf{att}}$-hybrid world are thus indistinguishable from those produced by the equivalent wrapped enclave in the $G_{\mathsf{att}}$-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 $G'_{\mathsf{att}}$ functionality. Honest parties in protocol $\mathcal{W}$ do not install any unwrapped program, and no external session will have direct access to $G_{\mathsf{att}}$ since it is installed as a $\mathcal{W}$ subroutine. If the (local) adversary attempts to install an unwrapped program to $G_{\mathsf{att}}$ directly, the simulator can run the program "in its head" without going through $G'_{\mathsf{att}}$, 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 $\mathrm{E}_i^L$, it generates a plausible enclave ID and returns it, without actually installing the enclave in $G_{\mathsf{att}}^{mod}$. While we do not explicitly define an enclave ID generation algorithm for $G_{\mathsf{att}}^{mod}$, 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 $\mathrm{E}_i^L$ 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 inter-

mediate 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 $G'_{\mathsf{att}}$. 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 $G_{\mathsf{att}}$, 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 $G^{mod}_{\mathsf{att}}$ 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 $G'_{\mathsf{att}}$ 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, 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 $G'_{\mathsf{att}}$ with a $G_{\mathsf{att}}$-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 $G'_{\mathsf{att}}$ shell. Indeed, the role of the $\mathcal{W}$ to $G'_{\mathsf{att}}$ simulator is to reconciling any such difference. We therefore have to analyse two distinctive cases.

**Theorem 4.** *Let $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ be any $G^{mod}_{\mathsf{att}}$ setups and a wrapper protocol such that Conjecture 1 holds, and additionally $G'_{\mathsf{att}}$ UC-emulates $\mathcal{W}$. For any protocol $\rho$ in the presence of $G'_{\mathsf{att}}$ that UC-emulates some $\mathcal{F}$ in the presence of $G'_{\mathsf{att}}$, $\rho$ in the presence of $\mathcal{W}$ UC-emulates $\mathcal{F}$ in the presence of $\mathcal{W}$.*

The statement follows from the composition theorem of [6, Theorem 3.3]. Showing that $G'_{\mathsf{att}}$ UC-emulates $\mathcal{W}$ (i.e. in conjunction with 1, $\mathcal{W}$ and $G'_{\mathsf{att}}$ 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 $G^{mod}_{\mathsf{att}}$ with the inverse transformation for attestation signatures described in the proof of 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 $\mathsf{prog}$. 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 $G'_{\mathsf{att}}$. 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 $G'_{\mathsf{att}}$ 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 5.** *Let $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ be any $G^{mod}_{\mathsf{att}}$ setups and a wrapper protocol such that Conjecture 1 holds for some simulator $S$. Let $G^S_{\mathsf{att}}$ be the combination of $G'_{\mathsf{att}}$ and $S$; for any protocol $\rho$ in the presence of $G^S_{\mathsf{att}}$ that UC-emulates some $\mathcal{F}$ in the presence of $G^S_{\mathsf{att}}$, $\rho$ in the presence of $\mathcal{W}$ UC-emulates $\mathcal{F}$ in the presence of $\mathcal{W}$.*

The statement above directly follows from [27, Lemma 1].

## 5.2 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 $G_{\mathsf{att}}$ as a subroutine and UC-realises $G'_{\mathsf{att}}$.
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 $\mathrm{W}^A[\mathsf{prog}, \mathrm{nextmsg}]$ never executes the internal protocol $\mathsf{prog}$ directly. Instead, the nextmsg() function now takes the code $\mathsf{prog}$ as an additional argument, and compiles it into the code for local enclaves $\mathrm{E}^L_{i\in\{1,\dots,n\}}$. When the party installs all enclaves $\mathrm{W}^A[\mathsf{prog}, \mathrm{nextmsg}], \mathrm{E}^L_1 \dots, \mathrm{E}^R_n\}$, it immediately resumes them with INIT, which allows the enclaves to conduce any necessary setup operations. Thereafter, on a resume call to $\mathsf{prog}$, $\mathrm{W}^A[\mathsf{prog}, \mathrm{nextmsg}]$ always begins its execution by running nextmsg first. When nextmsg returns $(\perp, \perp, v)$, 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 $G'_{\mathsf{att}}$, in favour of adopting a concrete example in Section 6. However, we do state the following for completeness:

*Conjecture 2.* Let $G_{\mathsf{att}} = G^{mod}_{\mathsf{att}}[\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}, \mathbb{S}], G'_{\mathsf{att}} = [\lambda, \mathsf{reg}, \mathbb{O}, \mathbb{A}', \mathbb{S}], \mathbb{A} \setminus \mathbb{A}' = A$. 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 $G_{\mathsf{att}}$ as a subroutine, UC-emulates $G'_{\mathsf{att}}$

**Theorem 6.** *Let $G_{\mathsf{att}}, G'_{\mathsf{att}}, \mathcal{W}$ be any $G^{mod}_{\mathsf{att}}$ setups and a wrapper protocol such that Conjecture 1 holds. For any protocol $\rho$ in the presence of $G'_{\mathsf{att}}$ that UC-emulates $\mathcal{F}$ in the presence of $G'_{\mathsf{att}}$, $\rho$ 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 $\rho$ 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 [6] 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 [6, Theorem 3.10], as the $\rho$ 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 $G'_{\mathsf{att}}$. 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 $G'_{\mathsf{att}}$ has been replaced with $\mathcal{W}$.

### 5.3 Interactions Between Features and Attacks

When defining the transformation between two versions of $G_{\mathsf{att}}$, 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. 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 $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ 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 $G_{\mathsf{att}}^{mod}$ with complex or unrealistic features.
Another illustrative example could be the introduction of cloning [59]. 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 $G_{\mathsf{att}}^{mod}$ oracle change (through feature addition and attack removal). Unlike the oracle shells in Section 4, 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, where they don't negatively interact with each other, to repeatedly apply Conjectures 1 and 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 $G'_{\mathsf{att}}$ to run in the $G_{\mathsf{att}}$-hybrid world where the oracle feature is not available, or be secure even if $G_{\mathsf{att}}$ allows an attack not in $G'_{\mathsf{att}}$. 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 [78, Table 1]).

## 6 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 in [13]. 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 [74]. 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, 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 5, that removes the Rollback interface from an ideal $G_{\mathsf{att}}^{mod}$ 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 $\text{Store}, \text{Fetch}$ oracles, why not just store the entirety of memory using this interface? There exist several protocols that claim to resolve rollback attacks (for example [67, 15, 72, 3, 54, 72, 39, 90])). We leave the formalisation of such a protocol to remove the adversarial rollback interface as future work. We believe that the current setting is still valuable, as it minimises the size of data stored in trusted memory (as well as providing an easy to explain protocol to prove an instance of Conjecture 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 $\text{nextmsg}_{\text{meas}}()$, which determines how to execute the wrapped program. When the enclave state is at the beginning of executing a RESUME instruction, $\text{nextmsg}_{\text{meas}}$ runs the MeasExec subroutine of $\text{W}^{A}[\cdot]$. 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, $\text{nextmsg}_{\text{meas}}$ 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.

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

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

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

**Fig. 7.** The $\text{W}^{A}[\cdot]$ enclave shell installed by protocol $\mathcal{W}$ for rollback iteration $c$ of enclave $\mathsf{eid}$ installed by party $\mathsf{pid}$ for session $\mathsf{idx}$

The code of the shell that implements the $\mathrm{W}^A[\cdot]$ program is presented in Figure 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 4.4 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 $\mathsf{prog}$. For most of the execution of $\mathsf{prog}$, 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 (as defined in Matetic et al. [67]) - 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 2 holds for the above protocol, we need to show that the protocol UC-emulates a $G'_{\mathsf{att}}$ 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 1, although we modify it appropriately for the adversarial case.

Assume the simulator has access to the same parameters as $\mathcal{W}$. The simulation translates all requests to install a wrapped enclave from corrupted parties into requests to install the unwrapped enclave in $G'_{\mathsf{att}}$; 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 $G'_{\mathsf{att}}$ functionality. During its execution, the shell $(\mathrm{sh}_{\mathbb{O},\mathbb{A}'}[\mathsf{prog}_{\mathsf{w}}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$ 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 measuring 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 $G'_{\mathsf{att}}$, and halt its own execution. Otherwise, if all the checks succeed, it returns the output value and attestation signatures produced by $G'_{\mathsf{att}}$. 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 $G_{\mathsf{att}}$). Attestation verification requests are forwarded to $G'_{\mathsf{att}}$ 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 $G'_{\mathsf{att}}$) 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 $G_{\mathsf{att}}$ subroutine - this is allowed by the identity bound. In that case, the simulator lets the message through to its local simulated $G_{\mathsf{att}}$ 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 $G^F_{\mathsf{att}}$. It is possible to think of $G^F_{\mathsf{att}}$ 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 $\mathsf{Steel}$ simulator of [13]. Alternatively, it is possible to see $G^F_{\mathsf{att}}$ as a bona-fide instance of $G^{mod}_{\mathsf{att}}$ 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 $\mathsf{Steel}$ simulator, in the presence of $G^{PST}_{\mathsf{att}}$, was required to keep a separate record of all messages signed by adversarial enclaves, this is the default for $G^{mod}_{\mathsf{att}}$, and therefore we do not require keeping track of any additional operations. $G^F_{\mathsf{att}}$ is taken to

be initialised with the same arguments as the real world $G_{\mathsf{att}}$ 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 $pk, s$
  $\mathrm{nextmsg_{meas}} \leftarrow \mathsf{map}^{\mathsf{L}}(\bar{\mathrm{E}}^R)$
  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 $\mathsf{eid}$
    $\mathrm{P}[P, \mathsf{idx}, \mathsf{eid}, \mathsf{prog_w}] \leftarrow (\emptyset, \emptyset)$
  else
    send $(\text{INSTALL}, \mathsf{prog})$ to $G^F_{\mathsf{att}}$ through $P$ and receive $\mathsf{eid}$
  return $\mathsf{eid}$
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 $\mathsf{input} \neq \epsilon$ then
      run $\mathsf{out}, \sigma \leftarrow \text{RESUME}(\mathsf{eid}, \mathsf{input}, \epsilon)$,
    else
      send $(\text{ITER}, c, i)$ to $\mathcal{A}$ on behalf of $(\mathrm{sh}_{\mathbb{O},\mathbb{A}}[\mathsf{prog}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
  else send $(\text{RESUME}, \mathsf{eid}, (i \parallel \mathsf{input}), \mathrm{Rollback})$ to $G^F_{\mathsf{att}}$ on behalf of $P$
On message $(\text{RESUME}, \mathsf{eid}, \cdot, \mathrm{Abort})$ from corrupted party $P$:
  if $(\cdot,\cdot) \in \mathrm{P}[P, \cdot, \mathsf{eid}, \cdot]$ then send $(\text{RESUME}, \mathsf{eid}, \epsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ on behalf of $P$
  else send $(\text{RESUME}, \mathsf{eid}, \epsilon, \mathrm{Abort})$ to $G^F_{\mathsf{att}}$ on behalf of $P$
On message $(\text{RESUME}, \mathsf{eid}, \mathsf{input}, a)$ from corrupted party $P$:
  if $(c, c_{latest}) \in \mathrm{P}[P, \cdot, \mathsf{eid}, \mathsf{prog_w}]$ then
    assert $a = \epsilon \vee a \in \mathbb{A}'$
    let $shEID \leftarrow (\mathrm{sh}_{\mathbb{O},\mathbb{A}'}[\mathsf{prog_w}], (\mathsf{eid}||\mathsf{pid}, \text{“att”}||\mathsf{idx}))$
    send FETCH to $\mathcal{A}$ through $shEID$ and receive $b$
    if $b \neq \mathrm{Continue} \vee c \neq c_{latest}$ then send $(\text{RESUME}, \mathsf{eid}, \epsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ and
  return
    send $(\text{RESUME}, \mathsf{eid}, \mathsf{input}, a)$ 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 RESPONSE
      send RESPONSE to $shEID$ on behalf of $\mathcal{A}$
      if RESPONSE= Abort then return
    receive $\mathsf{out}, \sigma$ from $G'_{\mathsf{att}}$
    send $(\text{STORE}, 1^{\mathbb{M}})$ to $\mathcal{A}$ through $shEID$ and receive $b'$
    if $b' \neq \mathrm{Continue}$ then send $(\text{RESUME}, \mathsf{eid}, \epsilon, \mathrm{Abort})$ to $G'_{\mathsf{att}}$ and return
    generate nonce $c' \xleftarrow{\$} \{0,1\}^\lambda, \mathrm{P}[P, \cdot, \mathsf{eid}, \mathsf{prog_w}] \leftarrow (c', c')$
    send $(\text{ITER}, c, c')$ to $\mathcal{A}$ on behalf of $shEID$
  else
    send $(\text{RESUME}, \mathsf{eid}, \mathsf{input}, \epsilon)$ to $G^F_{\mathsf{att}}$ and receive $\mathsf{out}, \sigma$
    return out, $\sigma$
  On message (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 (VERIFY, $\sigma, F(m, \mathrm{nextmsg}_\mathrm{meas})$) to $G'_\mathsf{att}$ and receive $v$
    else if $m$ is a measurement for a program prog with enclave ID eid installed by some
    party $P'$ in session idx, and $\mathrm{P}[P', \mathsf{idx}, \mathsf{eid}, \mathsf{prog} \neq \bot]$ then
        return $\bot$
    else
        send (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 $G'_\mathsf{att}$. If FETCH is allowed, the measurement stored will be the same as from the previous execution, and therefore the simulator runs the program in $G'_\mathsf{att}$. The behaviour of this execution is equivalent to the real world setup, since the shells of $G_\mathsf{att}$ and $G'_\mathsf{att}$ 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 $G_\mathsf{att}$ and $G'_\mathsf{att}$ 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[8]). The simulator perfectly reproduces this behaviour, by issuing an ABORT to the ideal enclave, after having issued the preceding FETCH.

## References

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

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

[3] 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`.

[4] 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`.

[8] 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.

[5] 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.

[6] 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.

[7] 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`.

[8] 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`.

[9] 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.

[10] Raad Bahmani et al. *Secure Multiparty Computation from SGX.* Cryptology ePrint Archive, Report 2016/1057. `https://eprint.iacr.org/2016/1057`. 2016.

[11] 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.

[12] Saskia Bayreuther et al. "Hidden $\Delta$-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`.

[13] 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`.

[14] 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.

[15] 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`.

[16] 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`.

[17] 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.

[18] Jan Camenisch et al. *iUC: Flexible Universal Composability Made Simple.* Cryptology ePrint Archive, Report 2019/1073. `https://eprint.iacr.org/2019/1073`. 2019.

[19] Ran Canetti. *Universally Composable Security: A New Paradigm for Cryptographic Protocols.* Cryptology ePrint Archive, Report 2000/067. `https://eprint.iacr.org/2000/067`. 2000.

[20] 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`.

[21] Ran Canetti. *Universally Composable Signatures, Certification and Authentication.* Cryptology ePrint Archive, Report 2003/239. `https://eprint.iacr.org/2003/239`. 2003.

[22] 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`.

[23] 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`.

[24] 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`.

[25] 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`.

[26] 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`.

[27] Ran Canetti et al. *Universally Composable End-to-End Secure Messaging.* Cryptology ePrint Archive, Report 2022/376. `https://eprint.iacr.org/2022/376`. 2022.

[28] 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`.

[29] 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`.

[30] 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`.

[31] 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.

[32] 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`.

[33] 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.

[34] 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.

[35] Michele Ciampi, Aggelos Kiayias, and Yu Shen. "Universal Composable Transaction Serialization with Order Fairness". In: *44th Annual International Cryptology Conference*. Lecture Notes in Computer Science. Aug. 2024.

[36] 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.

[37] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. https://eprint.iacr.org/2016/086. 2016.

[38] 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.

[39] 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/.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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/.

[45] 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.

[46] 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`.

[47] 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`.

[48] 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.

[49] Vipul Goyal et al. “Founding Cryptography on Tamper-Proof Hardware Tokens”. In: *TCC 2010: 7th Theory of Cryptography Conference*. Ed. by Daniele Micciancio. Vol. 5978. Lecture Notes in Computer Science. Zurich, Switzerland: Springer, Berlin, Heidelberg, Germany, Feb. 2010, pp. 308–326. DOI: `10.1007/978-3-642-11799-2_19`.

[50] 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.

[51] Dennis Hofheinz and Victor Shoup. *GNUC: A New Universal Composability Framework*. Cryptology ePrint Archive, Report 2011/303. `https://eprint.iacr.org/2011/303`. 2011.

[52] 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.

[53] 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`.

[54] 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`.

[55] 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`.

[56] Mahimna Kelkar et al. *Complete Knowledge: Preventing Encumbrance of Cryptographic Secrets*. Cryptology ePrint Archive, Report 2023/044. `https://eprint.iacr.org/2023/044`. 2023.

[57] 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`.

[58] 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.

[59] 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`.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] *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/.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] Stephan van Schaik et al. *SoK: SGX.Fail: How Stuff Get eXposed.* https://sgx.fail. 2022.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] Kevin R. B. Butler and Kurt Thomas, eds. *USENIX Security 2022: 31st USENIX Security Symposium.* Boston, MA, USA: USENIX Association, Aug. 2022.

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.