Steel: Composable Hardware-Based Stateful and Randomised Functional Encryption

Pramod Bhatotia[1,2], Markulf Kohlweiss[1], Lorenzo Martinico[1](✉), and Yiannis Tselekounis[1]

[1] School of Informatics, University of Edinburgh, Edinburgh, Scotland
{mkohlwei,lorenzo.martinico}@ed.ac.uk, tselekounis@sians.org
[2] Department of Informatics, TU Munich, Munich, Germany
pramod.bhatotia@in.tum.de

Abstract. Trusted execution environments (TEEs) enable secure execution of programs on untrusted hosts and cryptographically attest the correctness of outputs. As these are complex systems, it is essential to formally capture the exact security achieved by protocols employing TEEs, and ultimately, prove their security under composition, as TEEs are typically employed in multiple protocols, simultaneously.

Our contribution is twofold. On the one hand, we show that under existing definitions of attested execution setup, we can realise cryptographic functionalities that are unrealisable in the standard model. On the other hand, we extend the adversarial model to capture a broader class of *realistic adversaries*, we demonstrate weaknesses of existing security definitions this class, and we propose stronger ones.

Specifically, we first define a generalization of *Functional Encryption* that captures *Stateful and Randomised* functionalities (FESR). Then, assuming the ideal functionality for attested execution of Pass et al. (Eurocrypt '2017), we construct the associated protocol, Steel, and we prove that Steel UC-realises FESR in the *universal composition with global subroutines* model by Badertscher et al. (TCC '2020). Our work is also a validation of the compositionality of the Iron protocol by Fisch et al. (CCS '2017), capturing (non-stateful) hardware-based functional encryption.

As the existing functionality for attested execution of Pass et al. is too strong for real world use, we propose a weaker functionality that allows the adversary to conduct *rollback and forking attacks*. We demonstrate that Steel (realising stateful functionalities), contrary to the stateless variant corresponding to Iron, is not secure in this setting and discuss possible mitigation techniques.

1 Introduction

Due to the rise of cloud computing, most people living in countries with active digital economies can expect a significant amount of information about them to be stored on cloud platforms. Cloud computing offers economies of scale for computational resources with ease of management, elasticity, and fault tolerance driving

© International Association for Cryptologic Research 2021
J. A. Garay (Ed.): PKC 2021, LNCS 12711, pp. 709–736, 2021.
https://doi.org/10.1007/978-3-030-75248-4_25

further centralization. While cloud computing is ubiquitously employed for building modern online service, it also poses security and privacy risks. Cloud storage and computation are outside the control of the data owner and users currently have no mechanism to verify whether the third-party operator, even with good intentions, can handle their data with confidentiality and integrity guarantees.

Hardware-Based Solutions. To overcome these limitations, trusted execution environments (TEEs), such as Intel SGX [27], ARM Trustzone [45], RISC-V Keystone [29,38], AMD-SEV [33] provide an appealing way to build secure systems. TEEs provide a hardware-protected secure memory region called a *secure enclave* whose residing code and data are isolated from any layers in the software stack including the operating system and/or the hypervisor. In addition, TEEs offer remote attestation for proving their trustworthiness to third-parties. In particular, the remote attestation enables a remote party to verify that an enclave has a specific identity and is indeed running on a genuine TEE hardware platform. Given they promise a hardware-assisted secure abstraction, TEEs are now commercially offered by major cloud computing providers including Microsoft Azure [47], Google Cloud [46], and Alibaba Cloud [5].

Modeling Challenges. While TEEs provide a promising building block, it is not straightforward to design secure applications on top of TEEs. In particular applications face the following three challenges: (1) Most practical applications require combining trusted and untrusted components for improved performance and a low trusted computing base; (2) TEEs are designed to protect only the volatile, in-memory, "stateless" computations and data. Unfortunately, this abstraction is insufficient for most practical applications, which rely on stateful computation on untrusted storage mediums (SSDs, disks). Ensuring security for such untrusted storage mediums is challenging because TEEs are prone to rollback attacks; and lastly, (3) TEE hardware designs are prone to numerous side channel attacks exploiting memory access patterns, cache timing channels, etc. These side channel attacks have the potential to completely compromise the confidentiality, integrity, and authenticity (remote attestation) of enclaves.

Therefore, it is important to carefully model the security achieved by the protocols of such systems as well as the assumptions in the cryptography and the hardware, and the trust afforded in protocol participants. Ideally such modelling must be compositional to facilitate the construction of larger systems based on smaller hardware and cryptography components. Given a sufficiently expressive model of TEEs, they can be used as a powerful setup assumption to realise many protocols.

The model of Pass, Shi, and Tramer (PST) [44] takes an initial step towards modelling protocols employing TEEs. The PST model provides a compositional functionality for attested execution and shows how to instantiate various primitives impossible in the standard model, as well as some limitations of TEEs. The PST model was first weakened in [52], which provides a compelling example of how an excessively weak enclave, susceptible to side channel attacks that break confidentiality (but not integrity and authenticity), can still be used as setup for useful cryptographic primitives. Both models, however, live at two opposite extremes, and thus fail to capture realistic instantiations of real world trusted execution.

Functional Encryption and Limitations. One of the core primitives that enables privacy preserving computation and storage is *Functional Encryption* (FE), introduced by [16]. FE is a generalisation of Attribute/Identify Based Encryption [48,49], that enables authorized entities to compute over encrypted data, and learn the results in the clear. In particular, parties possessing the so-called functional key, sk_f, for the function f, can compute $f(x)$, where x is the plaintext, by applying the decryption algorithm on sk_f and an encryption of x. Access to the functional key is regulated by a trusted third party. While out of scope for our work, identifying such a party is an interesting question that requires establishing metrics for the trustworthiness of entities we might want to be able to decrypt functions, and the kind of functions that should be authorised for a given level of trust. An obvious option for the role of trusted authority would be that of a data protection authority, who can investigate the data protection practices of organisations and levy fines in case these are violated. Another approach could be decentralising this role, by allowing the functional key to be generated collectively by a number of data owners [1,23].

FE is a very powerful primitive but in practice highly non-trivial to construct. Motivated by the inefficiency of existing instantiations of FE for arbitrary functions, the work of [28] introduces Iron, which is a practically efficient protocol that realises FE based on Intel's SGX. In [28] the authors formally prove security of the proposed protocol, however their proof is in the standalone setting. In a related work, Matt and Maurer [41] show (building on [3]) that composable functional encryption (CFE) is impossible to achieve in the standard model, but achievable in the random oracle model. For another important variant of the primitive, namely, *randomized functional encryption*, existing constructions [2,30,37], are limited in the sense that they require a new functional key for each invocation of the function, i.e., decryptions with the same functional key always return the same output. Finally, existing notions of FE only capture *stateless* functionalities, which we believe further restricts the usefulness and applicability of the primitive. For instance, imagine a financial institution that sets its global lending rate based on the total liquidity of its members. Financial statements can be sent, encrypted, by each member, with each of these transactions updating the global view for the decryptor, who can then compute the function's result in real time.

Given the above limitations, in this work we leverage the power of hardware assisted computation to construct FE for a *broader class of functionalities* under the strongest notion of *composable security.*

1.1 Our Contributions

We consider a generalization of FE to arbitrary *stateful and probabilistic functionalities* (FESR), that subsumes multi-client FE [23] and enables cryptographic computations in a natural way, due to the availability of internal randomness. Our contributions are as follows:

- We formally define functional encryption for stateful and randomized functionalities (FESR), in the Universal Composition (UC) setting [28].
- We construct the protocol Steel and prove that it realizes FESR in the newly introduced Universal Composition with Global Subroutines (UCGS) model [9]. Our main building blocks are: (1) the functional encryption scheme of [28] and (2) the global attestation functionality of PST. Our treatment lifts the PST model to the UCGS setting, and by easily adapting our proofs one can also establish the UCGS-security of [28].
- Finally, we introduce a weaker functionality for attested execution in the UCGS model to allow rollback and forking attacks, and use it to demonstrate that Steel does not protect against these. Finally, we sketch possible mitigation techniques.

1.2 Technical Overview

Attested Execution via the Global Attestation Functionality G_{att} *of PST* [44]. Our UC protocols assume access to the *global attestation functionality*, G_{att}, that captures the core abstraction provided by a broad class of attested execution processors, such as Intel SGX [27]. It models multiple hardware-protected memory regions of a TEE, called *secure enclaves*. Each enclave contains trusted code and data. In combination with a call-gate mechanism to control entry and exit into the trusted execution environment, this guarantees that this memory can only be accessed by the enclave it belongs to, i.e., the enclave memory is protected from concurrent enclaves and other (privileged) code on the platform. TEE processing environments guarantee the authenticity, the integrity and the confidentiality of their executing code, data and runtime states, e.g. CPU registers, memory and others.

G_{att} is parametrised by a signature scheme and a registry that captures all the platforms that are equipped with an attested execution processor. At a high level, G_{att} allows parties to register programs and ask for evaluations over arbitrary inputs, while also receiving signatures that ensure correctness of the computation. Since the manufacturer's signing key pair can be used in multiple protocols simultaneously, G_{att} is defined as a *global functionality* that uses the same key pair across sessions.

Universal Composition with Global Subroutines [10]. In our work we model global information using the newly introduced UCGS framework, which resolves inconsistencies in GUC [19], an earlier work that aims to model executions in the presence of global functionalities. UCGS handles such executions via a *management* protocol, that combines the target protocol and one or more instances of the global functionality, and creates an embedding within the standard UC framework. In our work, G_{att} (cf. Sect. 2.2) is modeled as a global functionality in the UCGS framework (updating the original PST formulation in GUC).

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

Steel*: UCGS-secure FE for Stateful and Randomized Functionalities.* Steel is executed by the sets of parties discussed above, where besides encryptors, all other parties receive access to G_{att}, abstracting an execution in the presence of secure hardware enclaves. Our protocol is based on Iron [28], so we briefly revisit the main protocol operations: (1) **Setup**, executed by the trusted party C, installs a *key management enclave* (KME), running a program to generate *public-key encryption* and *digital signature*, key pairs. The public keys are published, while the equivalent secrets are kept encrypted in storage (using SGX's terminology, the memory is sealed). Each of the decryptors installs a *decryption enclave* (DE), and attests its authenticity to the KME to receive the secret key for the encryption scheme over a secure channel. (2) **KeyGen**, on input function F, calls KME, where the latter produces a signature on the measurement of an instantiated enclave that computes F. (3) When **Encrypt** is called by an encryptor, it uses the published public encryption key to encrypt a message and sends the ciphertext to the intended recipients. (4) **Decrypt** is executed by a decrypting party seeking to compute some function F on a ciphertext. This operation instantiates a matching function enclave (or resume an existing one), whose role is that of computing the functional decryption, if an authorised functional key is provided.

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

The statefulness of functional encryption is simply enabled by adding a state array to each functional enclave. The array is also stored locally by the corresponding decryption enclave, and is updated for every decryption of a given function. Similar to [44], a curious artefact in the protocol's modeling is the addition of a "backdoor" that programs the output of the function evaluation subroutine, such that, if a specific argument is set on the input, the function evaluation returns the value of that argument. The reason for this addition is to enable simulation of signatures over function evaluations that have already been computed using the ideal functionality. We note that this addition does not impact correctness, as the state array is not modified if the backdoor is used, nor confidentiality, since the output of this subroutine is never passed to any other party besides the caller B. Finally, a further addition is that our protocol requires the addition of a proof of plaintext knowledge on top of the underlying encryption scheme. The Steel protocol definition is presented in Sect. 4.

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

Our proof is via a sequence of hybrids in which we prove that the real world protocol execution w.r.t. Steel is indistinguishable from the ideal execution, in the presence of an ideal functionality that captures FE for stateful and randomized functionalities. The goal is to prove that the decryptor learns nothing more than an authorized function of the private input plaintext, thus our hybrids gradually fake all relevant information accessed by the adversary. In the first hybrid,[1] all signature verifications w.r.t. the attestation key are replaced by an idealized verification process, that only accepts message/signature pairs that have been computed honestly (i.e., we omit verification via Σ). Indistinguishability is proven via reduction to the EU-CMA security of Σ. Next we fake all ciphertexts exchanged between enclaves that carry the decryption key for the target ciphertext, over which the function is evaluated (those hybrids require reductions to the CCA security of PKE).[2] The next hybrid substitutes ZK proofs over the target plaintexts with simulated ones, and indistinguishability with the previous one reduces to the zero knowledge property of N. Then, for maliciously generated ciphertexts under PKE – which might result via tampering with honestly generated encryptors' ciphertexts – instead of using the decryption operation of PKE, our simulator recovers the corresponding plaintext using the extractability property of N. Finally, we fake all ciphertexts of PKE, that encrypt the inputs to the functions (this reduces to CPA security). Note that, in [28], the adversary outputs the target message, which is then being encrypted and used as a parameter to the ideal world functionality that is accessed by the simulator in a black box way. In this work, we consider a stronger setting in which the adversary directly outputs ciphertexts of it's choice. While in the classic setting for Functional Encryption (where Iron lives) simulation security is easily achieved by asking

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

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

the adversarial enclave to produce an evaluation for the challenge ciphertext, in FESR the simulator is required to conduct all decryptions through the ideal functionality, so that the decryptor's state for that function can be updated. We address the above challenge by using the extractability property of NIZKs: for maliciously generated ciphertexts our simulator extracts the original plaintext and ask the ideal FESR functionality for it's evaluation. Simulation-extractable NIZK can be efficiently instantiated, e.g., using zk-SNARKs [12]. Security of our protocol is formally proven in Sect. 5. The simulator therein provided could be easily adapted to show that the Iron protocol UCGS-realises Functional Encryption, by replacing the NIZK operations for maliciously generated ciphertexts with a decryption from the enclave, as described above.

Rollback and Forking Attacks. Modeling attested execution via G_{att} facilitates composable protocol design, however, such a functionality cannot be easily realized since real world adversaries can perform highly non-trivial *rollback* and *forking* attacks against hardware components. In Sect. 6, we define a weaker functionality for attested execution, called $G_{\mathsf{att}}^{\mathsf{rollback}}$, that aims to capture rollback and forking attacks. To achieve this, we replace the enclave storage array in G_{att} with a tree data structure. While the honest party only ever accesses the last leaf of the tree (equivalent to a linked list), a corrupt party is able to provide an arbitrary path within the tree. This allows them to rollback the enclave, by re-executing a previous (non-leaf) state, and to support multiple forks of the program by interactively selecting different sibling branches. We give an example FESR function where we can show that correctness does not hold if $G_{\mathsf{att}}^{\mathsf{rollback}}$ is used instead of G_{att} within Steel, and discuss how countermeasures from the rollback protection literature can be adopted to address these attacks, with a consideration on efficiency.

1.3 Related Work

Hardware is frequently used to improve performance or circumvent impossibility results, e.g. [4,26,42]. As a relevant example, Chung et al. [25] show how to use of stateless hardware tokens to implement functional encryption.

The use of attestation has been widely adopted in the design of computer systems to bootstrap security [43]. In addition to formalising attested execution, Pass, Shi and Tramer (PST) [44] show that 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. The PST model is the first work to formalise attested execution in the UC framework. The compositional aspect of UC allows for the reused of the model in several successive works [22,24,52,54]. Other attempts at providing a formal model for attested execution include the game-based models of Barbosa et al. [15], Bahmani et al. [13], Fisch et al. [28]. The latter model arises from the need to evaluate the security of Iron, a hardware-based realisation of functional encryption, which was later extended to verifiable functional encryption in Suzuki et al. [51].

Rollback attacks (also known as reset attacks in the cryptographic literature) are a common attack vectors against third-party untrusted computing infrastructure. An attacker who is in control of the underlying infrastructure can at times simply restart the system to restore a previous system state. Yilek [55] presents a general attack that is applicable to both virtual machine and enclave executions: it shows that an adversary capable of executing multiple rollback attacks on IND-CCA or IND-CPA secure encryption schemes might learn information about encrypted messages by running the encryption algorithm on multiple messages with the same randomness. In the absence of true hardware-based randomness that cannot be rolled back, these kinds of attacks can be mitigated using hedged encryption, a type of key-wrap scheme [32], such that for each encryption round, the original random coin and the plaintext are passed through a pseudorandom function to generate the randomness for the ciphertext.

The area of rollback attacks on TEEs is well studied. Platforms like SGX [21], TPMs [39], etc. provide trusted monotonic counters, from which it is possible to bootstrap rollback-resilient storage. However, trusted counters are too slow for most practical applications. Furthermore, they wear out after a short period of time. As their lifetime is limited, they are unreliable for applications that require frequent updates [40]. Moreover, an adversary that is aware of this vulnerability can attack protocols that rely exclusively on counters, by instantiating a malicious enclave on the same platform that artificially damages the counters.

To overcome the limitation of SGX counters, ROTE [40] uses a consensus protocol to build a distributed trusted counter service, with performance necessarily reduced through several rounds of network communication. In the same spirit, Ariadne [50] is an optimized (local) synchronous technique to increment the counter by a single bit flip for deterministic enclaves.

Speicher [14] and Palaemon [31] proposed an asynchronous trusted counter interface, which provide a systematic trade-off between performance and rollback protection, addressing some limitations of synchronous counters. The asynchronous counter is backed up by a synchronous counter interface with a period of vulnerability, where an adversary can rollback the state of a TEE-equipped storage server in a system until the last stable synchronous point. To protect against such attacks, these systems rely on the clients to keep the changes in their local cache until the counter stabilizes to the next synchronisation point.

Lightweight Collective Memory (Brandenburger et al. [17]) is a proposed framework that claims to achieve fork-linearizability: each honest client that communicates with a TEE (on an untrusted server that might be rolled back) can detect if the server is being inconsistent in their responses to any of the protocol clients (i.e. if they introduce any forks or non-linearity in their responses). Finally, [35,36,53], protect hardware memory against active attacks, while [6,34], protect cryptographic hardware against tampering and Trojan injection attacks, respectively.

2 Preliminaries

2.1 UC Background

Universal Composability (UC), introduced by Canetti [18], is a security framework that enables the security analysis of cryptographic protocols. It supports the setting where multiple instances of the *same*, or *different protocols*, can be executed concurrently. Many extensions and variants of the framework have been proposed over the years; our treatment is based on the recently released Universal Composability with Global Subroutines framework (UCGS) [10] and the 2020 version of UC [18]. We briefly summarise the aspects of UC and UCGS necessary to understand our work.

Universal Composability. Consider two systems of PPT interactive Turing machine instances $(\pi, \mathcal{A}, \mathcal{Z})$ and $(\phi, \mathcal{S}, \mathcal{Z})$, where $\mathcal{Z}$ is the initial instance, and $\pi, \mathcal{A}$ (and respectively $\phi, \mathcal{S}$) have comparable runtime balanced by the inputs of $\mathcal{Z}$. We say that the two systems are indistinguishable if $\mathcal{Z}$ making calls to $\pi, \mathcal{A}$ (resp. $\phi, \mathcal{S}$) cannot distinguish which system it is located in. The two systems are commonly referred to as the *real* and *ideal world* (respectively). $\mathcal{Z}$ can make calls to instances within the protocol by assuming the (external) identity of arbitrary instances (as defined by the control function). Depending on the protocol settings, it might be necessary to restrict the external identities available to the environment. A ξ-identity-bounded environment is limited to assume external identities as specified by ξ, a polynomial time boolean predicate on the current system configuration.

We now recall a few definitions. Please consult [10,18] or our full version for the formal definitions of terms such as *balanced*, *respecting*, *exposing*, *compliant.*

Definition 1 (UC emulation [18]). *Given two PPT protocols π, ϕ and some predicate ξ, we say that π UC-emulates ϕ with respect to ξ-identity bound environments (or π ξ-UC-emulates ϕ) if for any balanced ξ-identity-bounded environment and any PPT adversary, there exists a PPT simulator $\mathcal{S}$ such that the systems $(\phi, \mathcal{S}, \mathcal{Z})$ and $(\pi, \mathcal{A}, \mathcal{Z})$ are indistinguishable.*

Given a protocol π which UC-emulates a protocol ϕ, and a third protocol ρ, which calls ϕ as a subroutine, we can construct a protocol where all calls to ϕ are replaced with calls to π, denoted as $\rho^{\phi \rightarrow \pi}$.

Theorem 1 (Universal Composition [18]). *Given PPT protocols π, ϕ, ρ and predicate ξ, if π, ϕ are both subroutine respecting and subroutine exposing , ρ is (π, ϕ, ξ)-compliant and π ξ-UC-emulates ϕ, then protocol $\rho^{\phi \rightarrow \pi}$ UC-emulates ρ*

By the composition theorem, any protocol that leverages subroutine ϕ in its execution can now be instantiated using protocol π.

UCGS. As the name suggests, generalised UC (GUC) [19] is an important generalization of the UC model. It accounts for the existence of a shared subroutine γ, such that both ρ and its subroutine π (regardless of how many instances of π are called by ρ) can have γ as a subroutine. The presence of the *global subroutine* allows proving protocols that rely on some powerful functionality that needs to be globally accessible, such as a public key infrastructure (PKI) [20], a global clock [8], or trusted hardware [44].

Unfortunately GUC has inconsistencies and has not been updated from the 2007 to the 2020 version of UC.[3] Universal Composability with Global Subroutines [10] aims to rectify these issues by embedding UC emulation in the presence of a global protocol within the standard UC framework.

To achieve this, a protocol π with access to subroutine γ is replaced by a new structured protocol $\mu = M[\pi, \gamma]$, known as *management* protocol; μ allows multiplexing a single instance of π and γ into however many are required by ρ, by transforming the session and party identifiers. μ is a subroutine exposing protocol, and is given access to an execution graph directory instance, which tracks existing machines within the protocol, and the list of subroutine calls (implemented as a structured protocol). The execution graph directory can be queried by all instances within the extended session of μ, and is used to redirect the outputs of π and γ to the correct machine.

Below we revisit the UC emulation with global subroutines definition from [10].

Definition 2 (UC Emulation with Global Subroutines [10]). *Given protocols π, ϕ, and γ, π ξ-UC emulates ϕ in the presence of γ if $M[\pi, \gamma]$ ξ-UC emulates $M[\phi, \gamma]$*

Now we state the main UCGS theorem.

Theorem 2 (Universal Composition with Global Subroutines [10]). *Given subroutine-exposing protocols π, ϕ, ρ, and γ, if γ is a ϕ-regular setup and subroutine respecting, ϕ, π are γ-subroutine respecting, ρ is (π, ϕ, ξ)-compliant and $(\pi, M[x, \gamma], \xi)$-compliant for $x \in \{\phi, \pi\}$, then if π ξ-UC-emulates ϕ in the presence of γ, the protocol $\rho^{\phi \to \pi}$ UC-emulates ρ in the presence of γ.*

2.2 The G_{att} Functionality

We now reproduce the G_{att} global functionality defined in the PST model [44]. The functionality is parameterised with a signature scheme and a registry to capture all platforms with a TEE. The below functionality diverges from the original one in that we let vk be a global variable, accessible by enclave programs as $G_{\mathsf{att}}.\mathsf{vk}$. This allows us to use G_{att} for protocols where the enclave program does not trust the caller to its procedures to pass genuine inputs, making it necessary to conduct the verification of attestation from within the enclave.

[3] In a nutshell the inconsistency arises from a discrepancy in the proof that emulation for a single-challenge session version, called EUC (used to prove protocols secure), implies UC-emulation for the multi-challenge GUC notion (used to prove the composition theorem).

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

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

```
*On message* INITIALIZE *from a party* $P$*:*
    **let** $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda), \mathsf{vk} \leftarrow \mathsf{spk}, \mathsf{msk} \leftarrow \mathsf{ssk}$
*On message* GETPK *from a party* $P$*:*
    **return** $\mathsf{vk}$
*On message* (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 $\mathsf{eid} \in \{0,1\}^\lambda$, **store** $\mathcal{T}[\mathsf{eid}, P] = (\mathsf{idx}, \mathsf{prog}, \emptyset)$
    **send** $\mathsf{eid}$ to P
*On message* (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** $\mathcal{T}[\mathsf{eid}, P] = (\mathsf{idx}, \mathsf{prog}, \mathsf{mem}')$
    **let** $\sigma \leftarrow \Sigma.\mathsf{Sign}(\mathsf{msk}, (\mathsf{idx}, \mathsf{eid}, \mathsf{prog}, \mathsf{output}))$ and **send** $(\mathsf{output}, \sigma)$ **to** $P$
```

The G_{att} functionality is a generalisation over other TEE formalisations, such as the one in [28], which tries to closely model some SGX implementation details. For instance, their hardware primitive distinguishes between local and remote attestation by exposing two sets of functions to produce and verify *reports* (for local attestation) and *quotes* (for remote attestation). Both data structure include enclave metadata, a tag that can uniquely identify the running program, input and output to the computation and some authentication primitive based on the former (MAC for local reports, signature for remote quotes). The G_{att} primitive, intended as an abstraction over different vendor implementations, removes much of this detail: both local and remote attestation consist in verifying the output of a *resume* call to some enclave through a public verification key, available both to machines with and without enclave capabilities. The output of computations is similarly the (anonymous) id of the enclave, the UC session id, some unique encode for the code computed by the enclave (which could be its source code, or its hash), and the output of the computation. Unlike in the Iron model, input does not have to be included in the attested return value, but if security requires parties to verify input, the function ca return it as part of its output. On enclave installation, its memory contents are initialised by the specification of its code; this initial memory state is represented by symbol $\emptyset$.

3 Functional Encryption for Stateful and Randomized Functionalities

In this section we define the ideal functionality of functional encryption for *stateful* and *randomized functionalities* (FESR).

FESR *syntax.*

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

While the above definition matches with that of classical functional encryption, we inject non-determinism and statefulness (respectively) by adding two additional inputs to functions in the allowed function class

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

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

3.1 Properties of FESR

Matt and Maurer [41] shows that the notion of functional encryption is equivalent, up to assumed resources, to that of an access control (AC) repository, where some parties A are allowed to upload data, and other parties B are allowed to retrieve some function on that data, if they have received authorisation (granted by a party C). A party B does not learn anything else about the stored data, besides the function they are authorised to compute (and length leakage F_0).

To allow stateful and randomized functions, we extend the function class with support for private state and randomness as above. Whenever B accesses a function on the data from the repository, the repository draws fresh randomness, evaluates the function on the old state. The function updates the state and evaluates to a value. Intuitively, this ideal world AC repository models both confidentiality and correctness:

Confidentiality. Confidentiality holds as B does not learn anything about the data besides the evaluations of these stateful randomized functions.

Correctness. A stateful functionality defines a stateful automaton, a set of states $\mathcal{S}$, the initial state $\emptyset \in \mathcal{S}$, a probabilistic transition function $\delta : \mathcal{X} \times \mathcal{S} \rightarrow \mathcal{Y} \times \mathcal{S}$. For every transition, a new input is sampled from $\mathcal{R}$ and given to F along with the input, to determine the next state. The transition function determines, for a given input and the current state, the probability $\Pr_\delta$ that the automaton will find itself in a certain next state, as well as an output value. Correctness requires that all consecutive outputs must always be justified by some input and a state reachable via δ from $\emptyset$.

Correctness holds for the ideal world AC repository as B can make exactly those state transitions by accessing a function on the data from the repository.

3.2 UC Functionality

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

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

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

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

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

```
On message (SETUP, $P$) from party $\mathsf{C}$, for $P \in \{\mathsf{A}, \mathsf{B}\}$:
  $\mathsf{setup}[P] \leftarrow \mathsf{true}$
  send (SETUP, $P$) to $\mathcal{A}$
On message (SETUP, $P$) from $\mathcal{A}$, for $P \in \{\mathsf{A}, \mathsf{B}\}$:
  $\mathsf{setup}[P] \leftarrow \mathsf{true}$
```

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

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

Note that, our definition is along the lines of [11,41], however, as opposed to [11], in which A and/or C might also get corrupted, in this work we primarily focus on the security guarantees provided by FE, which is confidentiality of the encrypted message against malicious decryptors, B. Yet, it provides security against malicious encryptors, A, thus it satisfies *input consistency*, which was originally introduced by [11]. In addition, our definition is the first one that captures stateful and randomized functionalities, where the latter refers to the standard notion of randomized functionalities in which each invocation of the function uses independent randomness. Therefore, our protocol achieves a stronger notion of randomized FE than [2,30,37], which require a new functional key for each invocation of the function, i.e., decryptions with the same functional key always return the same output.

Both correctness and confidentiality clearly hold for the ideal functionality by inspection of the 4 lines $\mathrm{r} \leftarrow \mathcal{R}$, $\mathsf{s} \leftarrow \mathcal{P}[\mathsf{B}, \mathrm{F}]$, $(\mathrm{y}, \mathsf{s}') \leftarrow \mathrm{F}(\mathrm{x}, \mathsf{s}, \mathrm{r})$, and $\mathcal{P}[\mathsf{B}, \mathrm{F}] \leftarrow \mathsf{s}'$.

4 A UC-Formulation of Steel

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

- For simplicity of presentation, we assume a single instance each for A, B
- all communication between parties (α, β) occurs over secure channels $\mathcal{SC}_\alpha^\beta, \mathcal{SC}_\beta^\alpha$
- Functional keys are (attestation) signatures by an enclave $\mathsf{prog}_{\mathsf{KME}}$ on input (keygen, F) for some function F; it is easy, given a list of keys, to retrieve the one which authorises decryptions of F
- keyword **fetch** retrieves a stored variable from memory and aborts if the value is not found
- on keyword **assert** , the program checks that an expression is true, and proceeds to the next line, aborting otherwise
- all variables within an enclave are erased after use, unless saved to encrypted memory through the **store** keyword

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

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

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

```
Key Generation Authority C:
On message (SETUP, $P$):
  if $\mathsf{mpk} = \perp$ then
    $\mathsf{eid}_{\mathsf{KME}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{C.sid}, \mathsf{prog}_{\mathsf{KME}})$
    send GET to $\mathcal{CRS}$ and receive (CRS, $\mathsf{crs}$)
    $(\mathsf{mpk}, \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\text{init}, \mathsf{crs}, \mathsf{C.sid}))$
  if $P = \mathsf{A}$ then
    send (SETUP, $\mathsf{mpk}$) to $\mathcal{SC}_{\mathsf{A}}$
  else if $P = \mathsf{B}$ then
    send (SETUP, $\mathsf{mpk}, \mathsf{eid}_{\mathsf{KME}}$) to $\mathcal{SC}_{\mathsf{B}}$ and receive (PROVISION, $\sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}$)
    $(\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\text{provision}, (\sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD}, \mathsf{eid}_{\mathsf{KME}})))$
    send (PROVISION, $\mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{sk}}$) to $\mathcal{SC}_{\mathsf{B}}$
```

```
*On message* $(\textsc{Keygen}, \mathrm{F}, \mathsf{B})$*:*
    **assert** $\mathrm{F} \in \mathtt{F} \wedge \mathsf{mpk} \neq \bot$
    $((\mathrm{keygen}, \mathrm{F}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{KME}}, (\mathrm{keygen}, \mathrm{F}))$
    $\mathsf{sk}_{\mathrm{F}} \leftarrow \sigma$; **send** $(\textsc{keygen}, (\mathrm{F}, \mathsf{sk}_{\mathrm{F}}))$ **to** $\mathcal{SC}_{\mathsf{B}}$
```

Encryption Party A:

```
*On message* $(\textsc{Setup}, \mathsf{mpk})$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
    **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
    **store** $\mathsf{mpk}, \mathsf{crs}$; **return** $\textsc{setup}$
*On message* $(\textsc{encrypt}, \mathrm{m})$*:*
    **assert** $\mathsf{mpk} \neq \bot \wedge \mathrm{m} \in \mathcal{X}$
    $\mathsf{ct} \xleftarrow{r} \mathsf{PKE.Enc}(\mathsf{mpk}, \mathrm{m})$
    $\pi \leftarrow \mathcal{P}((\mathsf{mpk}, \mathsf{ct}), (\mathrm{m}, \mathrm{r}), \mathsf{crs}), \mathsf{ct}_{\mathsf{msg}} \leftarrow (\mathsf{ct}, \pi)$
    **send** $(\textsc{write}, \mathsf{ct}_{\mathsf{msg}})$ **to** $\mathcal{REP}$ and **receive** $\mathsf{h}$
    **return** $(\textsc{encrypted}, \mathsf{h})$
```

Decryption Party B:

```
*On message* $(\textsc{Setup}, \mathsf{mpk}, \mathsf{eid}_{\mathsf{KME}})$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
    **store** $\mathsf{mpk}; \mathsf{eid}_{\mathsf{DE}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog}_{\mathsf{DE}})$
    **send** $\textsc{Get}$ **to** $\mathcal{CRS}$ and **receive** $(\textsc{Crs}, \mathsf{crs})$
    $((\mathsf{pk}_{KD}, \cdot, \cdot), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, \text{init-setup}, \mathsf{eid}_{\mathsf{KME}}, \mathsf{crs}, \mathsf{B.sid})$
    **send** $(\textsc{provision}, \sigma, \mathsf{eid}_{\mathsf{DE}}, \mathsf{pk}_{KD})$ **to** $\mathcal{SC}_{\mathsf{C}}$ and **receive** $(\textsc{provision}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}})$
    $G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\text{complete-setup}, \mathsf{ct}_{\mathsf{key}}, \sigma_{\mathsf{KME}}))$
    **return** $\textsc{setup}$
*On message* $(\textsc{keygen}, (\mathrm{F}, \mathsf{sk}_{\mathrm{F}}))$ *from* $\mathcal{SC}^{\mathsf{C}}$*:*
    $\mathsf{eid}_{\mathrm{F}} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{B.sid}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}])$
    $(\mathsf{pk}_{\mathsf{FD}}, \sigma_{\mathrm{F}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{init}, \mathsf{mpk}, \mathsf{B.sid}))$
    $\mathcal{K}[\mathrm{F}] \leftarrow (\sigma_{\mathrm{F}}, \mathsf{eid}_{\mathrm{F}}, \mathsf{pk}_{\mathsf{FD}}, \mathsf{sk}_{\mathrm{F}})$
    **return** $(\textsc{assigned}, F)$
*On message* $(\textsc{decrypt}, \mathrm{F}, \mathsf{h})$*:*
    **assert** $\mathcal{K}[\mathrm{F}] \neq \bot$
    **send** $(\textsc{read}, \mathsf{h})$ **to** $\mathcal{REP}$ and **receive** $\mathsf{ct}_{\mathsf{msg}}$
    $(\sigma_{\mathsf{F}}, \mathsf{eid}_{\mathsf{F}}, \mathsf{pk}_{\mathsf{FD}}, \mathsf{sk}_{\mathrm{F}}) \leftarrow \mathcal{K}[\mathrm{F}]$
    $((\mathsf{ct}_{\mathsf{key}}, \mathsf{crs}), \sigma_{\mathsf{DE}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\mathrm{provision}, \sigma_{\mathsf{F}}, \mathsf{eid}_{\mathsf{F}}, \mathsf{pk}_{\mathsf{FD}}, \mathsf{sk}_{\mathrm{F}}, \mathrm{F}))$
    $((\mathrm{computed}, \mathrm{y}), \cdot) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \bot))$
    **return** $(\textsc{Decrypted}, \mathrm{y})$
```

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

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

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

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

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

5 UC-Security of Steel

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

The Identity Bound ξ on the Environment. Our restrictions are similar to [44], namely we assume that the environment can access G_{att} in the following ways: (1) Acting as a corrupt party, and (2) acting as an honest party but only for non-challenge protocol instances.

We now prove our main theorem.

Theorem 3 Steel *(Sect. 4) UC-realizes the* FESR *functionality (Sect. 3) in the presence of the global functionality* G_{att} *and local functionalities* $\mathcal{CRS}, \mathcal{REP}, \mathcal{SC}$, *with respect to the identity bound* ξ *defined above.*

We present a simulator algorithm such that, for all probabilistic adversaries running in polynomial time with the ability of corrupting B. Following [41], our proof considers static corruption of a single party B, we did, however, not encounter any road-blocks to adaptive corruption of multiple decryptors besides increased proof notational complexity. The environment is unable to distinguish between an execution of the Steel protocol in the real world, and the protocol consisting of $\mathcal{S}_{\text{FESR}}$, dummy parties A, C and ideal functionality FESR. Both protocols have access to the shared global subroutines of G_{att}. While hybrid functionalities $\mathcal{REP}$, $\mathcal{SC}$, $\mathcal{CRS}$ (for their definition, see the full version) are only available in the real world and need to be reproduced by the simulator, we use $\mathcal{SC}$ in the simulator to denote simulated channels, either between the simulator and corrupted parties (for corrupt parties), or between the simulator and itself (for honest parties).

Given protocols Steel, FESR, and G_{att}, Steel ξ-UC emulates FESR in the presence of G_{att} if $M[\mathsf{Steel}, G_{\mathsf{att}}]$ ξ-UC emulates $M[\text{FESR}, G_{\mathsf{att}}]$ (see Definition 2). We focus or exposition on the messages exchanged between the environment and the machine instances executing Steel, FESR, and G_{att}, since the machine M is simply routing messages; i.e., whenever $\mathcal{Z}$ wants to interact with the protocol, M simply forwards the message to the corresponding party; the same holds for G_{att}.

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

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

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

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

*On message* (KEYGEN, $F$, B) *from* FESR*:*
  assert $\mathrm{F} \in \mathsf{F} \wedge \mathsf{mpk} \neq \perp$
  $((\mathrm{keygen}, \mathrm{F}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid_{KME}}, (\mathrm{keygen}, \mathrm{F}))$
  $\mathsf{sk_F} \leftarrow \sigma$
  send (KEYGEN, $(\mathrm{F}, \mathsf{sk_F})$) to $\mathcal{SC}_{\mathsf{B}}$

**For Decryption Party B:**
*On message* GET *from party* B *to* $\mathcal{CRS}$*:*
  send (CRS, $\mathsf{crs}$) to B
*On message* (READ, $\mathsf{h}$) *from party* B *to* $\mathcal{REP}$*:*
  send (DECRYPT, $\mathrm{F}_0, \mathsf{h}$) to FESR on behalf of B and receive $|\mathrm{m}|$
  assert $|\mathrm{m}| \neq \perp$
  $\mathsf{ct} \leftarrow \mathsf{PKE.Enc}(\mathsf{mpk}, 0^{|\mathrm{m}|})$
  $\pi \leftarrow \mathsf{N}.\mathcal{S}_2(\mathsf{crs}, \tau, (\mathsf{mpk}, \mathsf{ct}))$
  $\mathsf{ct_{msg}} \leftarrow (\mathsf{ct}, \pi)$; $\mathcal{H}[\mathsf{ct_{msg}}] \leftarrow h$
  send (READ, $\mathsf{ct_{msg}}$) to B
*On message* (INSTALL, $\mathsf{idx}, \mathsf{prog}$) *from party* B *to* $G_{\mathsf{att}}$*:*
  $\mathsf{eid} \leftarrow G_{\mathsf{att}}.\mathsf{install}(\mathsf{idx}, \mathsf{prog})$
  $\mathcal{G}[\mathsf{eid}].\mathsf{install} \leftarrow (\mathsf{idx}, \mathsf{prog})$
  // $\mathcal{G}[\mathsf{eid}].install[1]$ is the program's code
  forward eid to B
*On message* (RESUME, $\mathsf{eid}, \mathsf{input}$) *from party* B *to* $G_{\mathsf{att}}$*:*
  // The $G_{\mathsf{att}}$ registry does not allow $B$ to access $\mathsf{eid_{KME}}$ in real
  world
  assert $\mathcal{G}[\mathsf{eid}] \neq \perp \wedge \mathsf{eid} \neq \mathsf{eid_{KME}}$
  if $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] \neq \mathsf{prog_{FE}}[\cdot] \vee \mathsf{input}[-1] \neq \perp$ then
      $(\mathsf{output}, \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}, \mathsf{input})$
      $\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \parallel (\sigma, \mathsf{input}, \mathsf{output}))$
      $\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], \mathsf{output})$
      if $\mathcal{G}[\mathsf{eid}].\mathsf{install}[1] = \mathsf{prog_{DE}} \wedge \mathsf{input}[0] = \mathrm{provision}$ then
          $(\mathrm{provision}, \sigma, \mathsf{eid}, \mathsf{pk_{FD}}, \mathsf{sk_F}, \mathrm{F}) \leftarrow \mathsf{input}$
          fetch $(\cdot, (\mathrm{init\text{-}setup}, \mathsf{eid_{KME}}, \mathsf{crs}), \cdot) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
          assert $(\mathsf{idx}, \mathsf{eid_{KME}}, \mathsf{prog_{KME}}, (\mathrm{keygen}, \mathrm{F})) \in \mathcal{B}[\mathsf{sk_F}]$
          assert $(\mathsf{idx}, \mathsf{eid_{DE}}, \mathsf{prog_{DE}}, \mathsf{ct_{key}}, \mathsf{crs}) \in \mathcal{B}[\sigma_{\mathsf{DE}}]$
      forward $(\mathsf{output}, \sigma)$ to B
  else
```

```
$\mathsf{idx}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}] \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{install}$
$(\mathrm{run}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathrm{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{msg}}, \mathsf{crs}, \bot) \leftarrow \mathsf{input}$
**assert** $(\sigma_{\mathrm{F}}, (\mathrm{init}), (\mathsf{pk}_{\mathsf{FD}})) \in \mathcal{G}[\mathsf{eid}].\mathsf{resume}$
**assert** $(\mathsf{idx}, \mathsf{eid}, \mathsf{prog}_{\mathsf{FE}}[\mathrm{F}], \mathsf{pk}_{\mathsf{FD}}) \in \mathcal{B}[\sigma_{\mathrm{F}}]$
**assert** $(\mathsf{idx}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{prog}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{crs})) \in \mathcal{B}[\sigma_{\mathrm{DE}}]$
// If the ciphertext was not computed honestly and saved to $\mathcal{H}$
**if** $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] = \bot$ **then**
    $(\mathsf{ct}, \pi) \leftarrow \mathsf{ct}_{\mathsf{msg}}$
    $(\mathrm{m}, \mathrm{r}) \leftarrow \mathsf{N}.\mathcal{E}(\tau, (\mathsf{mpk}, \mathsf{ct}), \pi)$
    **if** $\mathrm{m} = \bot$ **then send** $(\textsc{decrypt}, \mathrm{F}, \bot)$ **to** $\mathsf{B}$ and abort
    **send** $(\textsc{encrypt}, \mathrm{m})$ **to** FESR on behalf of $\mathsf{B}$ and **receive** $\mathsf{h}$
    $\mathcal{H}[\mathsf{ct}_{\mathsf{msg}}] \leftarrow \mathsf{h}$
$\mathsf{h} \leftarrow \mathcal{H}[\mathsf{ct}_{\mathsf{msg}}]$
**send** $(\textsc{decrypt}, \mathrm{F}, \mathsf{h})$ **to** FESR on behalf of $\mathsf{B}$ and **receive** $\mathrm{y}$
$((\mathrm{computed}, \mathrm{y}), \sigma) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \bot, \bot, \bot, \bot, \bot, \mathrm{y}))$
$\mathcal{G}[\mathsf{eid}].\mathsf{resume} \leftarrow \mathcal{G}[\mathsf{eid}].\mathsf{resume} \parallel (\sigma, \mathsf{input}, (\mathrm{computed}, \mathrm{y})))$
$\mathcal{B}[\sigma] \leftarrow (\mathcal{G}[\mathsf{eid}].\mathsf{install}[0], \mathsf{eid}, \mathcal{G}[\mathsf{eid}].\mathsf{install}[1], (\mathrm{computed}, \mathrm{y}))$
**forward** $((\mathrm{computed}, \mathrm{y}), \sigma)$ to $\mathsf{B}$
```

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

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

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

On message $(\textsc{keygen}, \mathrm{F}, B)$ from the functionality after a call to KEYGEN, $\mathcal{S}_{\mathrm{FESR}}$ simply produces a functional key by running the appropriate $\mathsf{prog}_{\mathsf{KME}}$ procedure through G_{att}. Similarly, on receiving $(\text{READ}, \mathsf{h})$ for $\mathcal{REP}$, $\mathcal{S}_{\text{FESR}}$ produces an encryption of a canonical message (a string of zeros) and simulates the response.

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

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

The full proof of security is available in the full version; for an overview, refer to Sect. 1.2.

6 Rollback and Forking Attacks

While the Attested Execution functionality modelled by G_{att} is a meaningful first step for modeling attested execution, it is easy to argue that it is not realisable (in a UC-emulation sense) by any of the existing Trusted Execution Environment platforms to date. In a follow-up paper, Tramer et al. [52] weaken the original G_{att} model to allow complete leakage of the memory state. 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. Additionally, there are many other classes of attacks that can not be expressed by this model. We now extend the G_{att} functionality to model *rollback* and *forking attacks* against an enclave.

6.1 $G_{\mathsf{att}}^{\mathsf{rollback}}$ Functionality

Our model of rollback and forking attacks is drawn from the formulation expressed in Matetic et al. [40], but with PST's improved modelling of attestation, which does not assume perfectly secure authenticated reads/writes between the attester and the enclave.

Matetic et al. model rollback by distinguishing between enclaves and enclave instances. Enclave instances have a distinct memory state, while sharing the same code. As with G_{att}, where the outside world has to call subroutines individually, the environment is not allowed to interact directly with a program once it is instantiated, except for pausing, resuming, or deleting enclave instances. Additionally, their model provides functions to store encrypted memory outside the enclave (*Seal*) and load memory back (*Unseal*).

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

In a forking attack an attacker manages to run two instances of the same enclave concurrently, such that, once the state of one instance is changed by an external operation, querying the other instance will result in an outdated state. This relies on both enclaves producing signature that at the minimum attest the same program. On a system where attestation uniquely identifies each copy of the enclave, a forking attack can still be launched by an attacker conducting multiple rollback attacks and feeding different stale snapshots to a single enclave copy [17].

Our new functionality $G_{\mathsf{att}}^{\mathsf{rollback}}$ employs this idea to model the effect of both rollback and forking attacks. We replace the internal mem variable of G_{att} with a tree data structure. The honest caller to the functionality will always continue execution from the memory state of an existing leaf of the tree while an adversary can specify an arbitrary node of the tree (through a unique node identifier), to which the state of the enclave gets reset. The output mem' will then be appended as a new child branch to the tree. To model a rollback attack, the adversary specifies the parent node for the next call to resume (or any ancestor node to execute a second rollback). To model a forking attack, the adversary can interactively choose nodes in different branches of the tree. The functionality is parameterised with a signature scheme and a registry to capture all platforms with a TEE, like in the original formulation.

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

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

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

The proposed rollback model is perhaps somewhat reductive, as it only allows "discrete" rollback operations, where memory states are quantised by program subroutines. It is conceivable that real world attackers would have a finer-grained rollback model, where they can interrupt the subroutine's execution, and resume from an arbitrary instruction.

Attack on Stateful Functional Encryption. Although our protocol uses probabilistic primitives, we deem the generic reset attack presented in [55] unrealistic for TEE platforms such as SGX, where an enclave is allowed direct access to a hardware-based source of randomness [7].

On the other hand, it easy to find a protocol-specific rollback attack on Steel. While F's state remains secret to a corrupt B interacting with $G_{\mathsf{att}}^{\mathsf{rollback}}$ (the memory is still sealed when stored), an adversary can make enclave calls produce results that would be impossible in the simpler model. As an example, take the following function from F that allows setting a key and sampling the output of a PRF function F for a single message:

```
function PRF-Wrapper($\mathrm{x}, \mathsf{mem}$)
    if $\mathsf{mem} = \emptyset$ then
        $\mathrm{K} \leftarrow \mathrm{x}$
        Store $\mathsf{mem} \leftarrow \mathrm{K}$
        return ACK
    else if $\mathsf{mem} = \vec{1}$ then
        return $\perp$
    else
        Store $\mathsf{mem} \leftarrow \vec{1}$
        return $\mathrm{F}_{\mathrm{K}}(\mathrm{x})$
```

An adversary who has completed initialisation of its decryption enclave with enclave id $\mathsf{eid}_{\mathsf{DE}}$, obtained a functional key sk through the execution of keygen on $\mathsf{eid}_{\mathsf{KME}}$, and initialised a functional enclave for PRF-Wrapper with enclave id $\mathsf{eid}_{\mathrm{F}}$, public key $\mathsf{pk}_{\mathsf{FD}}$ and attestation σ, executes the current operations for three ciphertexts $\mathsf{ct}_{\mathrm{K}}, \mathsf{ct}_{\mathrm{x}}, \mathsf{ct}_{\mathrm{x}'}$, encrypting a key K and plaintexts x, x':

```
1: $((\mathsf{ct}_{\mathsf{key}}, \mathsf{crs}), \sigma_{\mathsf{DE}}) \leftarrow G_{\mathsf{att}}.\mathsf{resume}(\mathsf{eid}_{\mathsf{DE}}, (\mathrm{provision}, \sigma, \mathsf{eid}, \mathsf{pk}_{\mathsf{FD}}, \mathsf{sk}))$
2: $((\mathrm{computed}, ACK), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{k}}, \mathsf{crs}, \perp), \mathsf{node})$
3: // node is the node id for a leaf for eid$_{\mathrm{F}}$'s mem tree
4: $((\mathrm{computed}, \mathrm{y}), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathrm{x}}, \mathsf{crs}, \perp), \mathsf{node}')$
```

```
5: // $\mathsf{node}'$ is the node id for a leaf for $\mathrm{eid}_{\mathrm{F}}$'s mem tree
6: $((\mathrm{computed}, \mathrm{y}'), \cdot) \leftarrow G_{\mathsf{att}}^{\mathsf{rollback}}.\mathsf{resume}(\mathsf{eid}_{\mathrm{F}}, (\mathrm{run}, \mathsf{vk}_{\mathsf{att}}, \sigma_{\mathsf{DE}}, \mathsf{eid}_{\mathsf{DE}}, \mathsf{ct}_{\mathsf{key}}, \mathsf{ct}_{\mathsf{x}'}, \mathsf{crs}, \perp),$
   $\mathsf{node}')$
7: // $\mathsf{node}'$ is the same node id as in the previous call (and thus to the
   parent of the current leaf in mem)
```

As a result of this execution trace, the adversary violates correctness by inserting an illegal transition (with input ϵ) in the stateful automaton for PRF-Wrapper, from state $\mathsf{access}(\mathsf{tree}, \mathsf{node}'.\mathit{child}) = \vec{1}$ back to $\mathsf{access}(\mathsf{tree}, \mathsf{node}') = [K]$, and then back to state $\vec{1}$ with input x'. The adversary can then obtain the illegal set of values $\mathrm{y} \leftarrow \mathrm{F}_{\mathrm{K}}(\mathrm{x})$ and $\mathrm{y}' \leftarrow \mathrm{F}_{\mathrm{K}}(\mathrm{x}')$, whereas in the ideal world after obtaining y, the only possible output for the function would be $\perp$ (the only legal transition from state $\vec{1}$ leads back to itself). The simulator is unable to address this attack, as the memory state is internal to the ideal functionality, and the key will always be erased after the second call.

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

Mitigation Techniques. In Sect. 1.3, we showed that rollback resilience for trusted execution environments is an active area of research, with many competing protocols. However, most solutions inevitably entail a performance trade-off.

Due to the modular nature of Steel, it is possible to minimise the performance impact. Observe that party B instantiates a single DE and multiple FE. We can reduce the performance penalty by making only DE rollback resilient. We guarantee correctness despite rollbacks of FE, by encoding a counter alongside the function state for each F. On a decryption request, the $\mathsf{prog}_{\mathsf{FE}}$ enclave is required to check in with the $\mathsf{prog}_{\mathsf{DE}}$ enclave to retrieve the decryption key as part of the provision call. To enable rollback resilience, we include the counter stored by $\mathsf{prog}_{\mathsf{FE}}$ as an additional parameter of this call. $\mathsf{prog}_{\mathsf{DE}}$ compares the counter received for the current evaluation of F with the one received during the last evaluation, and authorises the transfer of the secret key only if greater. Before evaluating the function, $\mathsf{prog}_{\mathsf{FE}}$ increases and stores its local counter.

To achieve rollback resilience for the $\mathsf{prog}_{\mathsf{DE}}$ enclave, we can rely on existing techniques in the literature, such as augmenting the enclave with asynchronous monotonic counters [14], or using protocols like LCM [17] or ROTE [40]. Formalising how these protocols can be combined with the $G_{\mathsf{att}}^{\mathsf{rollback}}$ functionality to achieve the fully secure G_{att} is left for future work.

We also note that Stateless functional encryption as implemented in IRON is resilient to rollback and forking because there is little state held between computation. Since we assume C is honest, the only programs liable to be attacked are DE and FE[F].

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

Acknowledgements. This research was partially supported by the National Cyber Security Centre, the UK Research Institute in Secure Hardware and Embedded Systems (RISE), and the European Union's Horizon 2020 Research and Innovation Programme under grant agreement 780108 (FENTEC).

References

1. Abdalla, M., Benhamouda, F., Kohlweiss, M., Waldner, H.: Decentralizing inner-product functional encryption. In: Lin, D., Sako, K. (eds.) PKC 2019. LNCS, vol. 11443, pp. 128–157. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17259-6_5
2. Agrawal, S., Wu, D.J.: Functional encryption: deterministic to randomized functions from simple assumptions. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10211, pp. 30–61. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56614-6_2
3. Agrawal, S., Gorbunov, S., Vaikuntanathan, V., Wee, H.: Functional encryption: new perspectives and lower bounds. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 500–518. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_28
4. Ahmad, A., Joe, B., Xiao, Y., Zhang, Y., Shin, I., Lee, B.: OBFUSCURO: a commodity obfuscation engine on intel SGX. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019. The Internet Society (2019). ISBN 1-891562-55-X. https://www.ndss-symposium.org/ndss-paper/obfuscuro-a-commodity-obfuscation-engine-on-intel-sgx/
5. Cloud, A.: TEE-based confidential computing. https://www.alibabacloud.com/help/doc-detail/164536.htm (2020)
6. Ateniese, G., Kiayias, A., Magri, B., Tselekounis, Y., Venturi, D.: Secure outsourcing of cryptographic circuits manufacturing. In: Baek, J., Susilo, W., Kim, J. (eds.) ProvSec 2018. LNCS, vol. 11192, pp. 75–93. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01446-9_5
7. Aumasson, J., Merino, L.: SGX secure enclaves in practice: security and crypto review. Black Hat **2016**, 10 (2016)
8. Badertscher, C., Maurer, U., Tschudi, D., Zikas, V.: Bitcoin as a transaction ledger: a composable treatment. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 324–356. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_11

9. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12552, pp. 1–30. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64381-2_1
10. Badertscher, C., Canetti, R., Hesse, J., Tackmann, B., Zikas, V.: Universal composition with global subroutines: capturing global setup within plain UC. In: Pass, R., Pietrzak, K. (eds.) TCC 2020. LNCS, vol. 12552, pp. 1–30. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64381-2_1
11. Badertscher, C., Kiayias, A., Kohlweiss, M., Waldner, H.: Consistency for functional encryption. Cryptology ePrint Archive, Report 2020/137 (2020). https://eprint.iacr.org/2020/137
12. Baghery, K., Kohlweiss, M., Siim, J., Volkhov, M.: Another look at extraction and randomization of groth's zk-SNARK. Cryptology ePrint Archive, Report 2020/811 (2020). https://eprint.iacr.org/2020/811
13. Bahmani, R., et al.: Secure multiparty computation from SGX. In: Kiayias, A. (ed.) FC 2017. LNCS, vol. 10322, pp. 477–497. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70972-7_27
14. Bailleu, M., Thalheim, J., Bhatotia, P., Fetzer, C., Honda, M., Vaswani, K.: SPEICHER: securing lsm-based key-value stores using shielded execution. In: Merchant, A., Weatherspoon, H. (eds.) 17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25–28, 2019, pages 173–190. USENIX Association (2019). URL https://www.usenix.org/conference/fast19/presentation/bailleu
15. Barbosa, M., Portela, B., Scerri, G., Warinschi, B.: Foundations of hardware-based attested computation and application to SGX. Cryptology ePrint Archive, Report 2016/014 (2016). http://eprint.iacr.org/2016/014
16. Boneh, D., Sahai, A., Waters, B.: Functional encryption: definitions and challenges. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 253–273. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19571-6_16
17. Brandenburger, M., Cachin, C., Lorenz, M., Kapitza, R.: Rollback and forking detection for trusted execution environments using lightweight collective memory. CoRR (2017). URL http://arxiv.org/abs/1701.00981v2
18. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000). http://eprint.iacr.org/2000/067
19. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Vadhan, S.P. (ed.) TCC 2007. LNCS, vol. 4392, pp. 61–85. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_4
20. Canetti, R., Shahaf, D., Vald, M.: Universally composable authentication and key-exchange with global PKI. In: Cheng, C.-M., Chung, K.-M., Persiano, G., Yang, B.-Y. (eds.) PKC 2016. LNCS, vol. 9615, pp. 265–296. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49387-8_11
21. Cen, S., Zhang, B.: Trusted time and monotonic counters with intel software guard extensions platform services (2017). https://software.intel.com/sites/default/files/managed/1b/a2/Intel-SGX-Platform-Services.pdf
22. Cheng, R., et al.: Ekiden: a platform for confidentiality-preserving, trustworthy, and performant smart contract execution. CoRR, abs/1804.05141 (2018). URL http://arxiv.org/abs/1804.05141

23. Chotard, J., Dufour Sans, E., Gay, R., Phan, D.H., Pointcheval, D.: Decentralized multi-client functional encryption for inner product. In: Peyrin, T., Galbraith, S. (eds.) ASIACRYPT 2018. LNCS, vol. 11273, pp. 703–732. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03329-3_24
24. Choudhuri, A.R., Green, M., Jain, A., Kaptchuk, G., Miers, I.: Fairness in an unfair world: fair multiparty computation from public bulletin boards. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS, Dallas, TX, USA, Oct. 31 - Nov. 2, 2017. pp. 719–728. ACM (2017)
25. Chung, K.-M., Katz, J., Zhou, H.-S.: Functional encryption from (small) hardware tokens. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013. LNCS, vol. 8270, pp. 120–139. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-42045-0_7
26. Ciampi, M., Lu, Y., Zikas, V.: Collusion-preserving computation without a mediator. Cryptology ePrint Archive, Report 2020/497 (2020). https://eprint.iacr.org/2020/497
27. Costan, V., Devadas, S.: Intel SGX explained. Cryptology ePrint Archive, Report 2016/086 (2016). http://eprint.iacr.org/2016/086
28. Fisch, B., Vinayagamurthy, D., Boneh, D., Gorbunov, S.: IRON: functional encryption using intel SGX. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS, Dallas, TX, USA, Oct. 31 - Nov. 2, 2017, pp. 765–782. ACM (2017)
29. Garlati, C., Pinto, S.: A clean slate approach to Linux security RISC-V enclaves (2020)
30. Goyal, V., Jain, A., Koppula, V., Sahai, A.: Functional encryption for randomized functionalities. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 325–351. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46497-7_13
31. Gregor, F., et al.: Trust management as a service: enabling trusted execution in the face of byzantine stakeholders. CoRR, abs/2003.14099 (2020). URL https://arxiv.org/abs/2003.14099
32. Hoang, V.T., Reyhanitabar, R., Rogaway, P., Vizár, D.: Online authenticated-encryption and its nonce-reuse misuse-resistance. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9215, pp. 493–517. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47989-6_24
33. Kaplan, D., Powell, J., Woller, T.: AMD memory encryption. White paper (2016)
34. Kiayias, A., Tselekounis, Y.: Tamper resilient circuits: the adversary at the gates. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013. LNCS, vol. 8270, pp. 161–180. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-42045-0_9
35. Kiayias, A., Liu, F.H., Tselekounis, Y.: Practical non-malleable codes from l-more extractable hash functions. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS, Vienna, Austria, Oct. 24–28, 2016. pp. 1317–1328. ACM (2016)
36. Kiayias, A., Liu, F.-H., Tselekounis, Y.: Non-malleable codes for partial functions with manipulation detection. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10993, pp. 577–607. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_20
37. Komargodski, I., Segev, G., Yogev, E.: Functional encryption for randomized functionalities in the private-key setting from minimal assumptions. J. Cryptol. **31**(1), 60–100 (2017). https://doi.org/10.1007/s00145-016-9250-8
38. Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: an open framework for architecting trusted execution environments. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16 (2020)

39. Levin, D., Douceur, J.R., Lorch, J.R., Moscibroda, T.: Trinc: small trusted hardware for large distributed systems. NSDI **9**, 1–14 (2009)
40. Matetic, S., et al.: ROTE: rollback protection for trusted execution. Cryptology ePrint Archive, Report 2017/048 (2017). http://eprint.iacr.org/2017/048
41. Matt, C., Maurer, U.: A definitional framework for functional encryption. In: Fournet, C., Hicks, M. (eds.) CSF 2015Computer Security Foundations Symposium, Verona, Italy, jul 13–17, pp. 217–231 IEEE (2015)
42. Nayak, K., et al.: HOP: hardware makes obfuscation practical. In: NDSS 2017, San Diego, CA, USA, Feb. 26 - Mar. 1, The Internet Society (2017)
43. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping trust in commodity computers. In: 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, May 16–19, pp. 414–429. IEEE Computer Society Press (2010)
44. Pass, R., Shi, E., Tramèr, F.: Formal abstractions for attested execution secure processors. In: Coron, J.-S., Nielsen, J.B. (eds.) EUROCRYPT 2017. LNCS, vol. 10210, pp. 260–289. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_10
45. Pinto, S., Santos, N.: Demystifying arm trustzone: a comprehensive survey. ACM Comput. Surv. **51**, 1–36 (2019)
46. Porter, N., Golanand, G., Lugani, S.: Introducing google cloud confidential computing with confidential VMs. (2020)
47. Russinovich, M.: Introducing azure confidential computing (2017)
48. Sahai, A., Waters, B.: Fuzzy identity-based encryption. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 457–473. Springer, Heidelberg (2005). https://doi.org/10.1007/11426639_27
49. Shamir, A.: Identity-based cryptosystems and signature schemes. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 47–53. Springer, Heidelberg (1985). https://doi.org/10.1007/3-540-39568-7_5
50. Strackx, R., Piessens, F.: Ariadne: a minimal approach to state continuity. In: Holz, T., Savage, S. (eds.) USENIX Security, Austin, TX, USA, Aug. 10–12, 2016, pp. 875–892. USENIX (2016)
51. Suzuki, T., Emura, K., Ohigashi, T., Omote, K.: Verifiable functional encryption using intel SGX. Cryptology ePrint Archive, Report 2020/1221 (2020). https://eprint.iacr.org/2020/1221
52. Tramer, F., Zhang, F., Lin, H., Hubaux, J. P., Juels, A., Shi, E.: Sealed-glass proofs: using transparent enclaves to prove and sell knowledge. Cryptology ePrint Archive, Report 2016/635 (2016). http://eprint.iacr.org/2016/635
53. Tselekounis, I.: Cryptographic techniques for hardware security. PhD thesis, University of Edinburgh, UK (2018). http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.763966
54. Wu, P., Shen, Q., Deng, R. H., Liu, X., Zhang, Y., Wu, Z.: ObliDC: an SGX-based oblivious distributed computing framework with formal proof. In: Galbraith, S.D., Russello, G., Susilo, W., Gollmann, D., Kirda, E., Liang, Z. (eds.) ASIACCS 19, Auckland, New Zealand, July 9–12, pp. 86–99. ACM (2019)
55. Yilek, S.: Resettable public-key encryption: how to encrypt on a virtual machine. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 41–56. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11925-5_4