Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers

Gabriel Kaptchuk
Johns Hopkins University
gkaptchuk@cs.jhu.edu

Ian Miers
Johns Hopkins University
imiers@cs.jhu.edu

Matthew Green
Johns Hopkins University
mgreen@cs.jhu.edu

Abstract

In this work we investigate the problem of achieving secure computation by combining stateless trusted devices with *public ledgers*. We consider a hybrid paradigm in which a client-side device (such as a co-processor or trusted enclave) performs secure computation, while interacting with a public ledger via a possibly malicious *host* computer. We explore both the constructive and potentially destructive implications of such systems. We first show that this combination allows for the construction of *stateful* interactive functionalities (including general computation) even when the device has no persistent storage; this allows us to build sophisticated applications using inexpensive trusted hardware or even pure cryptographic obfuscation techniques. We further show how to use this paradigm to achieve censorship-resistant communication with a network, even when network communications are mediated by a potentially malicious host. Finally we describe a number of practical applications that can be achieved today. These include the synchronization of private smart contracts; rate limited mandatory logging; strong encrypted backups from weak passwords; enforcing fairness in multi-party computation; and destructive applications such as *autonomous ransomware*, which allows for payments without an online party.

1 Introduction

Computer scientists have long sought to render computer systems immune to physical and logical attacks. Unfortunately, the high complexity of today's general-purpose computing systems has kept this goal far out of reach.

As a substitute, security engineers have focused on providing trustworthy computing for a limited portion of the computing base. This trend encompasses devices such as Hardware Security Modules, smart cards [Pou01], and "secure element" co-processors that have become ubiquitous in mobile devices [App16]. In a more general manifestation of this approach, manufacturers have begun to deploy Trusted Execution Environment (TEE) technology in commercial processors. This technology provides hardware and software support for isolated computing environments that run within a general-purpose computer. These environments (frequently called "enclaves") are exemplified by Intel's Software Guard Extensions (SGX) and ARM TrustZone [sgx, ARM17]. They are designed to remain secure even under conditions of total operating system compromise, and interact dynamically with the host system through a restricted interface.

While each of the examples above relies on hardware to achieve security, it is conceivable that future trusted enclaves may be implemented using pure software virtualization, general-purpose hardware obfuscation [GGH+13, DMMQN11, NFR+17], or even cryptographic program obfuscation [LMA+16]. While trusted execution environments have many applications in computing, they (and all secure co-processors) have fundamental limitations. Even if the trusted component ("enclave") operates perfectly, it remains dependent on a potentially malicious host computer for essential functions such as persistent data storage and network communication. This creates opportunities for a malicious host to manipulate the enclave and the enclave's view of the world. For example, such a host may:

1. Tamper with network communications, censoring certain inputs or outputs and preventing the enclave from communicating with the outside world.

2. Tamper with stored non-volatile data, *e.g.* replaying old stored state to the enclave in order to *reset* the state of a multi-step computation.

We stress that these attacks may have a catastrophic impact *even if the enclave itself operates exactly as designed*. For example, many interactive cryptographic protocols are vulnerable to "reset attacks", in which an attacker rewinds or resets the state of the computation [BFGM01, tpm, Gil15]. Moreover, state reset attacks are not merely a problem for cryptographic protocols: they also affect common applications such as limited-attempt password checking [Sko16].

When implemented in hardware, TEE systems such as Intel SGX can mitigate reset attacks by deploying a limited amount of tamper-resistant non-volatile storage.[1] However, such countermeasures are not available to environments written purely in software; those using fixed-key hardware obfuscation [NFR$^+$17]; or in instances where multiple copies of the same enclave run on different host machines, as they do in private smart contract systems [Hyp17]. Moreover, even hardware support cannot solve the problem of enforcing a secure channel to a public data network.

A hypothetical solution to these problems is to delegate statekeeping and network connectivity to a remote, trusted server or small cluster of peers, as discussed in [MAK$^+$17]. These could keep state on behalf of the enclave and would act as conduit to the public network. However, rather than solving the problem, this approach simply moves it to a different physical location, which is itself vulnerable to the same attacks. Moreover, provisioning and maintaining the availability of an appropriate server can be a challenge for many applications, including IoT deployments that frequently outlive the manufacturer.

Combining trusted enclaves with Ledgers. In this work we consider an alternative approach to ensuring the statefulness and connectivity of trusted computing devices. Unlike the strawman proposals above, our approach does not require the enclave to have secure internal non-volatile storage, nor does it require a protocol-aware external server to keep state.

Instead, we propose an alternative model in which parties have access to an append-only public ledger, or *bulletin board* with certain properties. Namely, upon publishing a string S on the ledger, a party receives a copy of the resulting ledger – or a portion of it – as well as a proof (*e.g.,* a signature) to establish that the publication occurred. Any party, including a trusted device, can verify this proof to confirm that the ledger data is authentic and was published on the real ledger. The main security requirement we require from the ledger is that its contents cannot be modified or erased, nor can a proof of publish be (efficiently) forged. This model has been previously investigated independently in other works, notably in the context of fair multiparty computation [CGJ$^+$17, GG17a].

Our use of this model is motivated by the fact that viable implementations of public ledgers are already available in practice. For example, our approach can be realized from existing centralized systems such as Google's Certificate Transparency project which logs issued certificates [cer18], or from distributed blockchain systems such as Bitcoin [Nak08] and Ethereum [eth] that use proof-of-work [GKW$^+$16] or proof-of-stake [KRDO16] to achieve consensus. Because the latter systems are distributed and have no single operator, they mitigate the risk of compromise that is inherent in any centralized system.

Our approach. In this work we propose a new general protocol, which we refer to as an *Enclave-Ledger Interaction* (ELI). This proposal divides any multi-step interactive computation into a protocol run between three parties: a stateless client-side *enclave* that contains a secret key; a *ledger* that logs posted strings and returns a proof of publication; and a (possibly adversarial) *host application* that facilitates all communications between the two preceding parties. Users may provide inputs to the computation via the host, or through the ledger itself. We illustrate our model in Figure 1.

We assume that the enclave is a trustworthy computing "device", such as a tamper-resistant hardware co-processor, TEE enclave or a cryptographically obfuscated circuit [NFR$^+$17, LMA$^+$16]. Most notably, we do not require the enclave to store persistent state, nor do we require it to possess a secure random number

[1]SGX provides approximately 200 monotonic counters implemented in internal NVRAM, and they have a limited update rate. Moreover, the literature affords many examples of attackers bypassing such mechanisms [SA03, Kau07, Sko16] using relatively inexpensive physical and electronic attacks.

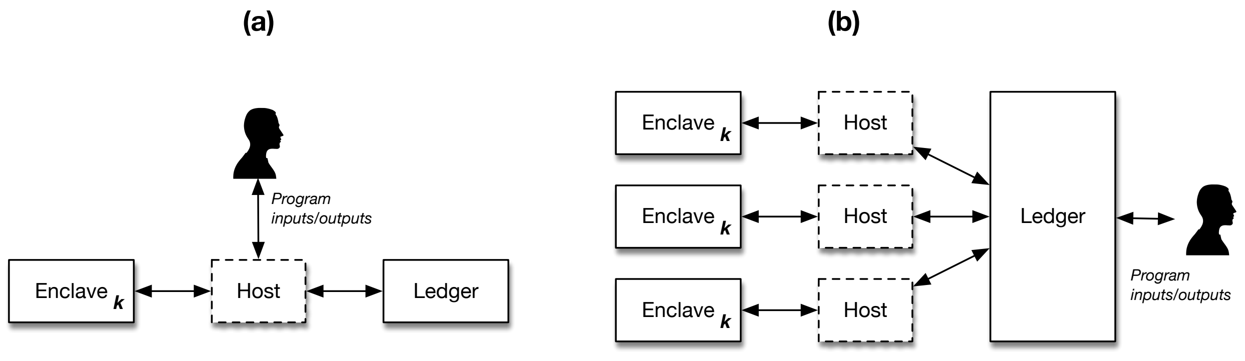

Figure 1: Two example ELI deployments. In the basic scenario (a) a single enclave (with a hard-coded secret key K) interacts with a ledger functionality via a (possibly adversarial) host application. Program inputs are provided by a user via the host machine. In scenario (b) multiple copies of the same enclave running on different host machines interact with the ledger (*e.g.,* as in a private smart contract system [Hyp17]), which allows them to synchronize a multi-step execution across many different machines without the need for direct communication. Program inputs and outputs may be provided via the ledger.

generator; we only require that the enclave possesses a single secret key K that is not known to any other party. We similarly require that the host application can publish strings to the ledger; access the ledger contents; and receive proofs of publication.

As a first contribution, we show how this paradigm can facilitate secure state management for randomized *multi-step* computations run by the enclave, even when the enclave has no persistent non-volatile storage or access to trustworthy randomness. Building such a protocol is non-trivial, as it requires simultaneously that the computation cannot be rewound or forked, even by an adversarial host application that controls all state and interaction with the ledger.

As a second contribution, we show that the combination of enclave plus ledger can achieve properties that may not be achievable even when the enclave uses stateful trusted hardware. In particular, we show how the combination of enclave plus ledger allows us to condition program execution on the *publication* of messages to the ledger, or the receipt of messages from third parties. For example, we can build enclaves that will not execute unless various outputs (or inputs) have been posted to the ledger.

As a third contribution, we describe several practical applications that leverage this paradigm. These include private smart contracts, limited-attempt password checking (which is known to be difficult to enforce without persistent state [Sko16]), enforced file access logging, and new forms of encryption that ensure all parties receive the plaintext, or that none do. As a practical matter, we demonstrate that on appropriate ledger systems that support payments, execution can be conditioned on other actions, such as *monetary payments* made to the ledger. In malicious hands, this raises the specter of *autonomous ransomware* that operates verifiably and without any need for a C&C or secret distribution center.

Concurrent work. Independently and concurrently with early drafts of this work [KMG17], other research efforts have considered the use of ledgers to achieve secure computation. Independently, Goyal and Goyal proposed the use of blockchains for implementing *one time programs* [GG17b] using cryptographic obfuscation techniques. While our work has a similar focus, we aim for a broader class of functionalities and a more practical set of applications. A collaboration of the current authors and Choudhuri *et al.* [CGJ$^+$17] use ledgers to obtain fairness for MPC protocols, an application that is discussed in later sections of this work. Finally, in unpublished work, Bowman et al. of Intel Corporation [Bow17] independently proposed some of the ideas in this work, and have begun to implement them in production smart contract systems. We believe Bowman's effort strongly motivates the formal analysis we include in this work. There have also been a number of attempts to combine trusted execution environments and public ledgers, but aimed at slightly different goals [KMS$^+$16, JKS16, ZCC$^+$16].

1.1 Intuition

We now briefly present the intuition behind our construction. Recall that our goal is to securely execute a *multi-step* interactive, probabilistic program $P : \mathcal{I} \times \mathcal{S} \to \mathcal{S} \times \mathcal{O}$, which we will define as having the following interface:

$$P(\mathsf{I}_i, \mathsf{S}_i; \bar{r}_i) \to (\mathsf{O}_i, \mathsf{S}_{i+1})$$

At each step of the program execution, the program takes a user input I_i, an (optional) state S_i from the previous execution step, along with some random coins $\bar{r}_i$. It produces an output O_i as well as an updated state S_{i+1} for subsequent steps. (Looking forward, we will add *public* ledger inputs and outputs to this interface as well, but we now omit these for purposes of exposition.) For this initial exposition, we will assume a simple ledger that, subsequent to each publication, returns the full ledger contents L along with a proof of publication σ. (In later sections we will discuss improvements that make this Ledger response *succinct*.)

The initial challenge in our setting is to execute this multi-step computation securely within an enclave that has only a single secret key K and no ability to persistently record state or communicate directly with the ledger. As illustrated in Figure 1 we assume that program inputs come from the host or the ledger.

We now discuss several candidate approaches, beginning with some of the ideas that are obviously insecure, and proceeding to a first version of our main construction.

Attempt #1: Encrypt program state. An obvious first step is for the enclave to simply encrypt each output state using its internal secret key, and to send the resulting ciphertext to the host for persistent storage. Assuming that we use a proper authenticated encryption scheme (and pad appropriately), this approach should guard both the confidentiality and authenticity of state values even when they are held by a malicious host.[2]

It is easy to see that while this prevents tampering with the contents of stored state, it does not prevent a malicious host from *replaying* old state ciphertexts to the enclave along with new inputs. In practice, such an attacker can rewind and fork execution of the program.

Attempt #2: Use the ledger to store state. A superficially appealing idea is to use the ledger itself to store an encrypted copy of the program state. As we will show, this does not mitigate rewinding attacks.

For example, consider the following strawman protocol: after the enclave executes the program P on some input, the enclave sends the resulting encrypted state to the ledger (via the host). The enclave can then condition the next execution of P on receiving valid ledger contents L, as well as a proof-of-publication σ, and then extracting the encrypted state from L.

Unfortunately this does nothing to solve the problem of adversarial replays. Because the enclave relies on the host to communicate with the ledger, a malicious host can simply replay old versions of L (including the associated proofs-of-publication) to the enclave, while specifying different program inputs. As before, this allows the host to fork the execution of the program.

Attempt #3: Bind program inputs on the ledger. To address the replay problem, we require a different approach. As in our first attempt, we will have the enclave send encrypted state to the host (and not the ledger) for persistent storage. As a further modification, we will add to this encrypted state an iteration counter i which identifies the next step of the program to be executed.

To execute the i^{th} invocation of the program, the host first commits its next program input I_i to the ledger. This can be done in plaintext, although for privacy we will use a secure commitment scheme. It labels the resulting commitment C with a unique identifier CID that identifies the enclave, and sends the pair (C, CID) to the ledger.

Following publication, the host can obtain a copy of the full ledger L as well as the proof-of-publication σ. It sends *all* of the above values (including the commitment randomness R) to the enclave, along with the most recent value of the encrypted state (or ε if this is the first step of the program). The enclave decrypts the encrypted state internally to obtain the program state and counter (S, i).[3] It verifies the following conditions:

[2]For the moment we will ignore the challenge of preventing re-use of nonces in the encryption scheme; these issues will need to be addressed in our main construction, however.

[3]If no encrypted state is provided, then i is implicitly set to 0 and $\mathsf{S} = \varepsilon$.

1. σ is a valid proof-of-publication for L.
2. The ledger L contains exactly i tuples $(\cdot, \mathsf{CID})$.
3. The most recent tuple embeds (C, CID).
4. C is a valid commitment to the input I using randomness R.

If all conditions are met, the enclave can now execute the program on state and input (S, I). Following execution, it encrypts the new output state and updated counter $(\mathsf{S}_{i+1}, i+1)$ and sends the resulting ciphertext to the host for storage.

Remark. Like our previous attempts, the protocol described above does *not* prevent the host from replaying old versions of L (along with the corresponding encrypted state). Indeed, such replays will still cause the enclave to execute P and produce an output. Rather, our purpose is to prevent the host from replaying old state with *different inputs.* By forcing the host to commit to its input on the ledger before L is obtained, we prevent a malicious host from changing its program input during a replay, (hopefully) ensuring that the host gains no new information from such attacks.

However, there remains a single vulnerability in the above construction that we must still address.

Attempt #4: Deriving randomness. While the protocol above prevents the attacker from changing the inputs provided to the program, there still remains a vector by which the malicious host could fork program execution. Specifically, even if the program input is fixed for a given execution step, the program execution may fork if the *random coins* provided to P change between replays. This might prove catastrophic for certain programs.[4]

To solve this problem, we make one final change to the construction of the enclave code. Specifically, we require that at each invocation of P, the enclave will derive the random coins used by the program in a deterministic manner. Fortunately, this can be done quite simply. For example, after verifying the checks described above, the enclave computes per-execution random coins as $\bar{r} \leftarrow \mathsf{PRF}_K(C \| i)$.[5] This approach fixes the random coins used at each computation step and effectively binds them to the ledger and the host's chosen input.

While the above construction provides an intuition for our ideas, our main construction includes a number of additional features and optimizations. We discuss several of these below.

Extension #1: Adding public input and output. A goal of our project is to allow the program to condition its execution on inputs and outputs drawn from (resp. sent to) the ledger. Fortunately, this task is made easier due the fact that at each execution of the above protocol the host provides the enclave with a complete and authenticated copy of L. This allows the enclave (and P) to condition its operation on *e.g.,* messages or public payment data found on the ledger.

To enforce public output, the program P can output a "public output string" as part of its output to the host, and can record this string within its encrypted state. By structuring the enclave code (or P) appropriately, the program can *require* the host to post this string to the ledger as a condition of further program execution. Of course, this is not an absolute guarantee that the host will publish the output string. That is, the enclave cannot force the host to post such messages. Rather, we achieve a best-possible guarantee in this setting: the enclave can simply disallow *further* execution in the host does not comply with the protocol.

Extension #2: Specifying the program. In the pedagogical presentation above, the program P is assumed to be fixed within the enclave. As a final extension, we note that the enclave can be configured to provide an environment for running arbitrary programs P, which can be provided as a separate input at each call. Achieving this involves recording (a hash) of P within the encrypted state, although the actual construction requires some additional checks to allow for a security proof. We include this capability in our main construction.

[4]For example, many interactive identification and oblivious RAM protocols become insecure if the program can be rewound and executed different randomness.

[5]In later sections of this paper we will consider a more robust approach that bases the random coins used at step i on all previous steps in the computation. This prevents certain attacks that occur if the host can *forge* a small number of ledger proofs.

Extension #3: Reducing Ledger bandwidth. As discussed above, the pedagogical protocol above requires the host and enclave to parse *the entire ledger* L on each execution step. This is obviously undesirable, as it requires the Enclave to (potentially) parse large amounts of data at each execution step. Moreover, it exposes the system to denial of service attacks in the event that an attacker can inject large amounts of superfluous data onto the Ledger. In our main construction we show that these issues can be addressed if the Ledger is given modest additional capabilities: namely (1) the ability to organize posted data into sequences (or chains), where each posted string contains a unique pointer to the preceding post, and (2) the ability for the Ledger to calculate a collision-resistant hash chain over these sequences. As we discuss in §2.2 and §5.2, these capabilities are provided by many candidates Ledger systems such as public (and private) blockchain networks.

Limitations of our pedagogical construction. The construction above is intended to provide an intuition, but is not the final protocol we describe in this work. An astute reader will note that this pedagogical example has many limitations, which must be addressed in order to derive a practical ELI protocol.

Finally, there exist subtle vulnerabilities in settings where an attacker can *forge* a small number of ledger proofs, as may be the case in proof-of-work blockchains where unforgeability guarantees are essentially economic in nature. Our main construction achieves these guarantees.

1.2 Applications

To motivate our techniques, we describe a number of practical applications that can be implemented using the ELI paradigm, including both constructive and potentially destructive techniques.

Synchronizing private smart contracts. Smart contract systems [Eth17, Hyp17] comprise a set of volunteer nodes that interact with a ledger to perform a multi-step interactive computation (a "contract"). In these systems, each computing step is typically performed by a different node, while the remaining nodes verify the correctness of resulting state transition as recorded on the ledger. Some production systems [Hyp17] have recently proposed to use TEEs to execute *private* contracts, where individual nodes (with pre-positioned shared secrets) perform the computation within an enclave, and output an *attestation* to the correctness of the output. A key challenge in these systems is the need to synchronize state as the computation migrates from node to node, and to ensure that each step correctly updates the correct ledger state. Motivated by an independent effort of Bowman *et al.* [Bow17] we show that our ELI paradigm achieves the necessary guarantees for security in this setting.

Limiting password guessing. Many cryptographic access control systems employ passwords to control access to encrypted filesystems [App16, Pro17] and cloud backup images (*e.g.,* Apple's iCloud Keychain [Krs16]). This creates a tension between the requirement to support easily memorable passwords (such as device PINs) while simultaneously preventing attackers from simply *guessing* users' relatively weak passwords [Bon12, USB$^+$15].[6] One approach to addressing this is to incorporate tamper-resistant hardware such as onboard co-processors [App16, Pro17, ARM17] and Hardware Security Modules [Krs16]. However, these systems are expensive (particularly in the online backup case [Krs16]) and may fail catastrophically when an attacker can rewind state.[7] We show that using our paradigm we can safely enforce *passcode guessing limits* even when using inexpensive hardware that cannot guarantee immutable state [Sko16].

Mandatory logging for local file access. In some corporate and enterprise settings, clients must publish access logs for sensitive files. This requires that each file access be recorded by some online system. We propose to use the ELI protocol to *mandate* logging of each file access before the necessary keys for an encrypted file can be accessed by the user. In this setting, log entries may be public (*i.e.,* visible to all parties on the network) or they may be encrypted to a specific management authority.

[6] This is made more challenging due to the fact that manufacturers have begun to design systems that do not include a trusted party – due to concerns that trusted escrow parties may be compelled to unlock devices [App17].

[7] See [Sko16] for an example of how such systems can be defeated when state is recorded in standard NAND hardware, rather than full tamper-resistant hardware.

Fairness. Choudhuri *et al.* [CGJ+17] proposed the use of ledgers as a way to ensure fairness in multi-party computation (MPC) protocols.[8] Their technique is to *encrypt* the output of an MPC execution, and require parties to post this ciphertext to a ledger before it can be decrypted. We first show that this technique can be constructed from an ELI, and further generalize the technique to construct *fair encryption*, that ensures a ciphertext can be decrypted *if and only if it has been published on a ledger*.

Autonomous ransomware. Ransomware, first formally examined in [YY96], is a class of malware that encrypts a victim's files, then demands a monetary ransom in exchange for decryption. Modern ransomware platforms are tightly integrated with cryptocurrencies such as Bitcoin, which act as both the ransom currency and a communication channel to the attacker [Sin16]. Once a system has been infected, users must transmit an encrypted key package along with a ransom payment to the attacker, who responds with the necessary decryption keys. The need to deliver secret keys is a fortunate weakness in the current ransomware paradigm, as it is both costly and increases the probability that she will be traced by law enforcement [Goo13]. More critically, this makes ransom payment a risky proposition for the victim, who cannot verify *a priori* that paying the ransom will restore access to data.[9] In this work we note a potentially *destructive* application of the ELI paradigm: the creation of ransomware that operates autonomously – from infection to decryption – with no need for remote parties to deliver secret keys. This ransomware employs local trusted hardware or obfuscation to store a decryption key for a user's data, and conditions decryption of a user's software on payments made on a public consensus network.

2 Definitions

Protocol Parties. A Enclave-Ledger Interaction is a protocol between three parties: the enclave functionality $\mathcal{E}$, the ledger $\mathcal{L}$, and a host application $\mathcal{H}$. We now describe the operation of these components.

The ledger $\mathcal{L}$. The ledger functionality provides a public append-only ledger for storing certain public data. Our key requirement is that the ledger is capable of producing a publicly-verifiable authentication tag σ_i over the entire ledger contents, or a portion of the ledger.

The enclave $\mathcal{E}$. The trusted enclave models a cryptographic obfuscation system or a trusted hardware co-processor configured with an internal secret key K. The enclave may contain the program P, or this program may be provided to it by the host application. Each time the enclave is invoked by the host application $\mathcal{H}$, it calculates and returns data to the host.

The host application $\mathcal{H}$. The host application is a (possibly adversarial) party that invokes both the enclave and the ledger functionalities. The host determines the inputs to each round of computation – perhaps after interacting with a user – and receives the outputs of the computation from the enclave.

2.1 The Program Model

Our goal in an ELI is to execute a multi-step interactive computation that runs on inputs that may be chosen *adaptively* by an adversary. We define this program as a single function $P : \mathcal{I} \times \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{O} \times \mathcal{P} \times \mathcal{S}$ that has the following input/output interface:

$P(\mathsf{I}_i, \mathsf{S}_i; \bar{r}_i) \rightarrow (\mathsf{O}_i, \mathsf{Pub}_i, \mathsf{S}_{i+1})$. On input a user input I_i, the current program state S_i, and random coins $\bar{r}_i$, this algorithm produces a program output O_i, as well as an optional public broadcast message Pub_i and new state S_{i+1}.

[8]This technique was in fact first proposed in an earlier public draft of this manuscript, but was developed more thoroughly in [CGJ+17].

[9]Indeed an emerging class of *pseudo-ransomware* has exploited this flaw to extort money without actually providing the ability to decrypt the locked files (in some cases unintentionally [Tre16b, Cim15, Tre16a]).

In our main construction, we will allow the host application to specify the program P that the enclave will run. This is useful in settings such as smart contract execution, where a given enclave may execute multiple distinct smart contract programs. As a result of this change, we will assume that P is passed as input to each invocation of the enclave.

Maximum program state size and runtime We assume in this work that the runtime of each P can be upper bounded by a polynomial function of the security parameter. We also require that for any program P used in our system there exists an efficiently-computable function $\mathsf{Max}(\cdot)$ such that $\mathsf{Max}(P)$ indicates the *maximum* length in bits of any output state S_i produced by P, and that $\mathsf{Max}(P)$ is polynomial in the security parameter.

One-Time vs. Multi-Use Programs In this work we consider two different classes of program. While all of our programs may involve multiple execution steps, *one-time programs* can be initiated only once by a given enclave. Once such a program has begun its first step of execution, it can never be restarted. By contrast, *multi-use programs* can be executed as many times as the user wishes, and different executions may be interwoven. However each execution of the program is independent of the others, receives different random coins and holds different state. In our model, an execution of a program will be uniquely identifier by a session identifier, which we denote by CID. Thus, the main difference between a one-time and many-time program is whether the enclave will permit the re-initiation of a given program P under a different identifier CID.

We note that it is possible to convert any multi-time program to a one-time program by having the enclave generate the value CID deterministically from its internal key K and the program P (*e.g.,* by calculating a pseudorandom function on these values), and then to enforce that each execution of the program P is associated with the generated CID. This enforcement algorithm can be instantiated as a "meta-program" P' that takes as input a second program P and is executed using our unmodified ELI protocol.

While our pedagogical example in the introduction discussed one-time programs, in the remainder of this work we will focus on multi-use programs, as these are generally sufficient for our proposed applications in §4.

2.2 Modeling the Ledger

The ledger models a public append-only bulletin board that allows parties to publish arbitrary strings. On publishing a string S to the ledger, all parties obtain the published string (and perhaps the full ledger contents) as well as a publicly-verifiable *authentication tag* to establish the string was indeed published. For security, we require that the authentication tags follow a standard notion of unforgeability, which we will discuss below.

In our pedagogical example earlier in this paper, we employed a naïve model of the ledger in which the enclave obtained *the entire* ledger contents L along with a publicly verifiable "authentication tag" σ subsequent to every post. However, relying on this form of ledger interaction can be quite bandwidth-intensive for the parties involved, as the ledger may incorporate posts from many different users. In our main constructions we will assume a ledger with some enhanced capabilities, including the ability to reference specific chains of posts made as part of a related execution, and to compute a collision-resistant hash chain over the posted strings. (Later in this section we will demonstrate that this interface can be constructed locally given access to a naïve ledger that returns the full ledger contents. As such we are not truly adding new requirements.)

The Ledger Interface Our model will assume an ideal ledger that posts an arbitrary strings S as part of specific a *chain* of posted values. For the purposes of our Ledger abstraction, we will require each post to identify a *chain identifier* CID. While the host may generate many such identifiers (and thus create an arbitrary number of distinct chains), in our abstraction will assume that other parties (*e.g.,* other host machines) will not be allowed to post under the same identifier. (The exact nature of the identifier CID depends on the specific Ledger instantiation, which we discuss in detail in §5.2).

The advantage of our interface is that similar checks and hash calculations are natural properties to achieve when using blockchain-based consensus systems to instantiate the ledger, since many consensus systems can perform the necessary checks as part of their consensus logic. Indeed, we note that many real-world ledger systems, including Bitcoin and Ethereum, provide these capabilities already, as we discuss in §5.2. By using these implementations, we can significantly reduce the cost of deploying our system.

We now define our ledger abstraction, which has the following interface.[10] Let $\mathsf{H_L}$ be a collision-resistant hash function:

- $\mathsf{Ledger.Post}(\mathsf{Data}, \mathsf{CID}) \rightarrow (\mathsf{post}, \sigma)$.

  When a party wishes to post a string Data onto the chain identified by CID, the ledger constructs a data structure post by performing the following steps:

  1. It finds the most recent $\mathsf{post_{prev}}$ on the Ledger that is associated with CID (if one exists).
  2. If $\mathsf{post_{prev}}$ was found, it sets $\mathsf{post.PrevHash} \leftarrow \mathsf{post_{prev}.Hash}$.
     Otherwise it sets $\mathsf{post.PrevHash} \leftarrow$ (**Root:** CID), where this labeling uniquely identifies it as first post associated with CID.
  3. It sets $\mathsf{post.Data} \leftarrow \mathsf{Data}$.
  4. It sets $\mathsf{post.Hash} \leftarrow \mathsf{H_L}(\mathsf{post.Data} \| \mathsf{post.PrevHash})$.
  5. It records $(\mathsf{post}, \mathsf{CID})$ on the public ledger.

  The ledger computes an authentication tag σ over the entire structure post and returns $(\mathsf{post}_, \sigma)$.

- $\mathsf{Ledger.Verify}(\mathsf{post}, \sigma) \rightarrow \{0, 1\}$.

  The verify algorithm is a public algorithm that will return 1 only if the authenticator σ was authentically generated by the ledger over that specific post. In general, this can be viewed as analogous to the verification algorithm of a digital signature scheme.[11]

For some applications it may also be desirable for third parties to possess an interface to read data from the ledger. We omit this interface for simpicity of exposition, although we stress that the ledger is *public* and hence such a functionality is implicit.

Remark. We note that the functionality of the above ledger can be simulated locally by an enclave that receives the *full* ledger contents L. Specifically, on receiving the full contents of a ledger L can construct our abstraction by *e.g.,* setting CID to be the public key of a digital signature scheme, and signing each message; it can then scan the full ledger to compute Hash_{i-1} and Hash_i locally.

Security of the Ledger. Informally, we require that it is difficult to construct a new pair (S, σ) such that $\mathsf{Ledger.Verify}(S, \sigma) = 1$ except as the result of a call to $\mathsf{Ledger.Post}$, even after the adversary has received many authenticator values on chosen strings. Intuitively we refer to this definition as SUF-AUTH. This definition is analogous to the SUF-CMA definition used for signatures.[12] We note that in our proofs we will assume an oracle that produces authentication tags, optionally without actually posting strings to a real ledger. For example, in a ledger based on signatures, our proofs might assume the existence of a signing oracle that produces signatures on chosen messages.

Concrete realizations of the Ledger. In Section 5.2 we discuss several concrete realizations of this ledger abstraction, using Certificate Transparency, Bitcoin-style networks, smart contract systems, and private blockchains.

[10]We omit the ledger *setup* algorithm for this description, although many practical instantiations will include some form of setup or key generation.

[11]We require that it is difficult to construct a pair (S, σ) such that $\mathsf{Ledger.Verify}(S, \sigma) = 1$ except as the result of a call to $\mathsf{Ledger.Post}$. Intuitively we refer to this definition as SUF-AUTH. This definition is analogous to the SUF-CMA definition used for signatures. Indeed, looking forward, these signatures will be one of the techniques used to construct the authentication tag.

[12]Indeed, in many practical instantiations of ledgers, these signatures will be one of the techniques used to construct the authentication tag.

$\mathsf{Setup}(1^\lambda) \to (K, \mathsf{pp})$. This trusted setup algorithm is executed once to configure the enclave. On input a security parameter λ, it samples a long-term secret K which is stored securely within the enclave, and the (non-secret) parameters pp which are provided to the enclave and the host.

$\mathsf{ExecuteApplication}(\mathsf{pp}, P)$**.** This algorithm is run on the host. It proceeds in an infinite loop, invoking the ledger operations and enclave operations. In each iteration of the loop, the user selects a step input, commits to it and posts it to the ledger. It then sends that input and the Ledger's output into the enclave to actually execute the next step.

$\mathsf{ExecuteEnclave}_{K,\mathsf{pp}}((P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)) \to (\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_i)$. This algorithm is run by the enclave, which is configured with K, pp. At the i^{th} computation step it takes as input a program P, an encrypted previous state $\mathcal{S}_i$, a program input I_i, commitment randomness r_i, a ledger output post_i and a ledger authentication tag σ_i. The enclave invokes P and produces a public output O_i, as well as a new encrypted state $\mathcal{S}_{i+1}$ and a public output Pub_i.

Figure 2: Definition of an Enclave Ledger Interaction (ELI) scheme.

2.3 Enclave-Ledger Interaction

An ELI scheme consists of a tuple of possibly probabilistic algorithms (Setup, $\mathsf{ExecuteEnclave}$, $\mathsf{ExecuteApplication}$). The interface for these algorithms is given in Figure 2.

2.4 Correctness and Security

Correctness. Correctness for an ELI scheme is defined in terms of the program P. Intuitively, at each step of execution, an honest enclave (operating in combination with the an honest host and ledger) should correctly evaluate the program P on the given inputs.

2.4.1 Simulation Security

We define security for a Enclave-Ledger interaction using a real/ideal-world definition. We formalize this definition below.

Intuition. Our main definition of ELI security considers the resilience of the protocol against a malicious host that controls the communication between an honest enclave and ledger, and can invoke each party on chosen messages. This definition is simulation-based, and requires us to define the following two experiments. In the first experiment, which we term the **Real** experiment, we consider an interaction where an adversarial host user $\mathcal{H}$ interacts with an honest ledger and honest enclave to execute the ELI protocol. These interfaces are provided to the host in the form of oracles that the host may call repeatedly.

In the **Ideal** experiment, we consider an adversarial *ideal* host $\hat{\mathcal{H}}$ that interacts with a trusted functionality. At each step of the experiment, this functionality takes as input a program, a program input, and a "session ID" provided by $\hat{\mathcal{H}}$, and runs the program using real random coins and with the most recent program state it has associated with this session ID. The trusted functionality stores the resulting state internally, records the public outputs on a table available to all parties, and returns both outputs to the user. This ideal model intuitively describes what we wish to accomplish from a secure multi-step interactive computing system.

Our security definition requires that for every p.p.t. adversarial real-world hosts $\mathcal{H}$, there must exist a p.p.t. ideal-world host $\hat{\mathcal{H}}$ that does "as well" in the ideal experiment as the real host does in the real experiment. More formally, we define this last notion via an indistinguishability-based definition: we require that the output distributions of $\mathcal{H}$ and $\hat{\mathcal{H}}$ are computationally indistinguishable between the two experiments.

What if the ledger is forgeable? In some of our proposed instantiations, the ledger authentication tags are *economically* secure, rather than cryptographically secure. In this setting we must contemplate the possibility that an attacker might be able to forge a small number of authentication tags, at a high cost.

Naturally such forgeries will result in a deviation from our ideal protocol. However, it is reasonable to consider how security *degrades* in the face of these forgeries.

For example, it is easy to imagine some ELI protocol that operates securely when forgeries are infeasible, but that becomes catastrophically insecure when the enclave is presented with a *single* forged authentication tag. For example, a contrived ELI protocol might be constructed to reveal a share of the secret K with each output, such that a single adversarial rewind allows the adversary to obtain K. This would clearly be catastrophic for the security of an ELI! While we cannot prevent an attacker from obtaining some advantage using forgeries, we wish to more clearly bound the attacker's capability in this setting.

Our approach to this problem is to extend both the **Real** and **Ideal** experiments with an additional capability. In the **Real** experiment, the adversary $\mathcal{H}$ is permitted to query a **Forgery** oracle that, on input a string S, provides a valid pair (S, σ) but *does not* update the ledger. In the **Ideal** experiment we provide a **Fork** oracle that allows the ideal adversary $\hat{\mathcal{H}}$ to run a single step of the computation using an older state (of their choosing). In each experiment, the adversaries are restricted to querying these respective oracles a maximum of q_{forge} times, where q_{forge} is a parameter provided as input to the experiment. To model the special case where the ledger is not forgeable, we can simply set $q_{\mathsf{forge}} = 0$.

Intuitively, this extended definition models the ability of an attacker to rewind a single step of the computation (per forgery) but without giving this attacker the ability to catastrophically break the security of the system, or even to run additional *legitimate* execution steps from this execution step. We stress that this definition represents a form of *best possible* security in our model, when forgeries are allowed.

Formal definitions. We now formally describe the experiments and present our definition of security.

The **Real** *experiment.* The real-world experiment is parameterized by a security parameter λ, an adversarial host $\mathcal{H}$ and a non-negative integer q_{forge}. At the start of the experiment, compute $(K, \mathsf{pp}) \leftarrow \mathsf{Setup}(1^\lambda)$ and output pp to $\mathcal{H}$. Subsequently, the adversary $\mathcal{H}$ is given access to three oracles: an *enclave* oracle; a *ledger* oracle, and a *forgery* oracle. The adversary can make an unlimited number of calls to the first two oracles, but is restricted to making at most q_{forge} queries to the forgery oracle. The oracles operate as follows:

The Ledger Oracle. The ledger oracle honestly implements the ledger interface used in the real protocol (see §2.2).

The Enclave Oracle. The enclave oracle is initialized with the secret K and parameters pp. Each time the user executes this oracle on some input $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)$, the oracle runs $\mathsf{ExecuteEnclave}_{K,\mathsf{pp}}(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)$. It returns the resulting output to $\mathcal{H}$.

The Forgery Oracle. This special oracle allows up to q_{forge} queries by $\mathcal{H}$. On input a string S, the forgery oracle computes a valid ledger authenticator σ over S *but does not place this data on the ledger.* It returns (S, σ).

At the conclusion of the experiment, the adversary produces an output. This output is the result of the experiment.

The **Ideal** *experiment.* This experiment is parameterized by a security parameter λ, an ideal-world adversarial host $\hat{\mathcal{H}}$ and a non-negative integer q_{forge}. The adversary is given access to two oracles: a *Compute* oracle and a *Fork* oracle. The oracles share a single table that is initially empty, and records tuples of the following structure:

$$(j, P_j, l \in \{\mathsf{compute}, \mathsf{fork}\}, \mathsf{S}_j, \mathsf{CID})$$

For $i = 1$ to q, the adversary $\hat{\mathcal{H}}$ calls the oracles as follows.

The Compute Oracle. On input $(P_i, \mathsf{I}_i, \mathsf{CID})$, the oracle first checks an internal table to find the most recent entry labeled with $\mathsf{compute}$ and CID. If one is found, it obtains the tuple $(j, P_j, \mathsf{compute}, \mathsf{S}_j, \mathsf{CID})$ where j has the highest value of all matching tuples in the table. If $P_j \neq P_i$ it aborts. If no tuple is found, it sets $\mathsf{S}_j \leftarrow \varepsilon$. Now it samples the coins $\bar{r}_i$ uniformly at random and computes:

$$(\mathsf{O}, \mathsf{Pub}, \mathsf{S}') \leftarrow P_i(\mathsf{I}_i, \mathsf{S}_j; \bar{r}_i)$$

The oracle stores $(i, P_i, \mathsf{compute}, \mathsf{S}', \mathsf{CID})$ in its internal table, places Pub on a list of "public outputs" and returns $(\mathsf{O}, \mathsf{Pub})$ to the caller.

The Fork Oracle. This oracle may be called by $\hat{\mathcal{H}}$ at most q_{forge} times. It operates similarly to the **Compute** oracle, except that it allows the caller execute the program on any *any past state* recorded by either oracle. When $\hat{\mathcal{H}}$ makes the i^{th} query on input $(P_i, \mathsf{I}_i, \mathsf{CID}, j)$, this oracle first looks up (j, CID) in the table to obtain the tuple $(j, P_j, l \in \{\mathsf{compute}, \mathsf{fork}\}, \mathsf{S}_j, \mathsf{CID})$ in the **Compute** oracle's internal table. If no match is found, or if $P_i \neq P_j$ it aborts. It now samples random coins $\bar{r}_i$ and computes:

$$(\mathsf{O}, \mathsf{Pub}, \mathsf{S}') \leftarrow P_i(\mathsf{I}_i, \mathsf{S}_j; \bar{r}_i)$$

The oracle now stores $(i, P_i, \mathsf{fork}, \mathsf{S}', \mathsf{CID})$ in its internal table. It returns $(\mathsf{Pub}, \mathsf{O})$ to the caller.

At the conclusion of the experiment, the adversary $\hat{\mathcal{H}}$ produces an output, which is the output of the experiment.

With these considerations in mind, we now present our main security definition:

Definition 2.1 (Simulation security for ELI) An ELI scheme $\Pi = (\mathsf{ExecuteApplication}, \mathsf{ExecuteEnclave})$ is *simulation-secure* if for every *p.p.t.* adversary $\mathcal{H}$, sufficiently large λ, and non-negative q_{forge}, there exists a p.p.t. $\hat{\mathcal{H}}$ such that the following holds:

$$\mathbf{Real}(\mathcal{H}, \lambda, q_{\mathsf{forge}}) \overset{c}{\approx} \mathbf{Ideal}(\hat{\mathcal{H}}, \lambda, q_{\mathsf{forge}})$$

Discussion. The **Forgery** and **Fork** oracles in the experiments above have similar purposes. Each is designed to mimic the degradation in security that occurs in the event of a ledger forgery. If $q_{\mathsf{forge}} = 0$, neither oracle may ever be called; this allows us to model the case where the ledger operates perfectly. If $q_{\mathsf{forge}} > 0$ then we wish our ELI protocol to minimize the damage caused by such forgeries. In the ideal world, we define that damage as *the ability to execute one additional computation* on pre-existing state, without updating the current state (or producing a public output). An important implication of this definition is that the resulting output states produced by the **Fork** oracle may *not* be fed to the **Compute** oracle. This prevents the attacker from spawning an entirely new execution branch from a single forgery.

2.4.2 Third Party Privacy

Our main security definition above considers the integrity of ELI computations in the face of an adversarial host application. However, because an ELI ledger involves an interaction with a *public* ledger, this raises an additional privacy concern: it is possible that a third party may try to recover sensitive, non-public program input by observing the ledger. To this end we propose a second security definition that ensures this leakage is limited only to the public information that such an eavesdropper will inevitably receive – namely, public program outputs and the existence of transactions.

Intuitively, we capture this definition by requiring that for every set of (adaptively chosen) program inputs given to an honest host and enclave, an adversary who sees the ledger output learns essentially "no more" information than is revealed by the explicit public outputs of the program. In practice we address this property via a second real/ideal world definition:

In the *real-world experiment*, the adversarial third-party takes on the role of an honest-but-curious ledger $\mathcal{L}$. It is provided with an oracle that allows it to request that the host run an instance of a chosen program P on a given input and identity CID. The host then runs the normal ELI protocol using the ledger and enclave to compute an output. The adversary receives all data posted to the ledger.

```
Algorithm 1: Setup
  Data: Input: $1^\lambda$
  Result: Secret $K$ for the enclave and public commitment parameters $\mathsf{pp}$
  $K \xleftarrow{\$} \{0,1\}^\lambda$
  $\mathsf{pp} \leftarrow \mathsf{CSetup}(1^\lambda)$
  Output $(K, \mathsf{pp})$
```

```
Algorithm 2: ExecuteApplication
  Data: Input: $\mathsf{pp}, P$
  // Set the initial counter to 0 and the initial state to $\varepsilon$
  $\mathcal{S}_0 \leftarrow \varepsilon$
  $i \leftarrow 0$
  $\mathsf{Pub}_0 \leftarrow \varepsilon$
  // Loop and run the program
  while true do
      Obtain $\mathsf{I}_i$ from the user
      if $\mathsf{I}_i = \bot$ then
          Terminate
      $r_i \xleftarrow{\$} \{0,1\}^\ell$
      $C_i \leftarrow \mathsf{Commit}(\mathsf{pp}, (i, \mathsf{I}_i, \mathcal{S}_i, P); r_i)$
      $(\sigma_i, \mathsf{post}_i) \leftarrow \mathsf{Ledger.Post}((\mathsf{Pub}_i, C_i))$
      $(\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_i) \leftarrow \mathsf{ExecuteEnclave}(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)$
      Output $(\mathsf{O}_i, \mathsf{Pub}_i)$ to the user
      $i \leftarrow i + 1$
```

Figure 3: Setup and ExecuteApplication algorithms for our main construction.

In the *ideal-world experiment*, we assume a trusted party that receives the same inputs directly from the adversary, and executes the program internally – producing only the public output (or ε if there is none). Our security notion is captured as follows: we require that for all real-world adversaries $\mathcal{L}$ (playing the real experiment), there must exist an ideal-world adversary $\hat{\mathcal{L}}$ (playing the ideal experiment) such that the output distributions of $\hat{\mathcal{L}}$ and $\mathcal{L}$ are computationally indistinguishable.

We formalize this definition in Appendix A.

3 Our Construction

In this section we present a specific construction of an Enclave-Ledger Interaction scheme. Our construction makes black box use of commitment schemes, authenticated symmetric encryption, collision-resistant hash functions and pseudorandom functions.

Notation: In our constructions we define $\mathsf{Verify}(\cdot)$ as a primitive that verifies a statement, and aborts the program (with output $\bot$) if the statement evaluates to false. Let $(\mathsf{Pad}, \mathsf{Unpad})$ be a padding algorithm that, on input a program P, pads a series of inputs to the maximum length of the provided data.

Commitment schemes. Let $\Sigma_{\mathsf{com}} = (\mathsf{CSetup}, \mathsf{Commit})$ be a commitment scheme where CSetup generates public parameters pp. The algorithm $\mathsf{Commit}(\mathsf{pp}, M; r)$ takes in the public parameters, a message M, along with random coins r, and outputs a commitment $C \neq \varepsilon$ which can be verified by re-computing the commitment on the same message and coins.

Algorithm 3: ExecuteEnclave

```
Data: Input: $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)$
        Internal values: $K, \mathsf{pp}$
Result: $(\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_i)$ or $\perp$
// Verify and parse the provided inputs
Assert ($\mathsf{Ledger.Verify}(\mathsf{post}_i, \sigma_i)$)
Assert($\mathsf{post}_i.\mathsf{Hash} = \mathsf{H_L}(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash})$)
Parse $(\mathsf{Pub}_{i-1}, C_i) \leftarrow \mathsf{post.Data}$
Assert($C_i = \mathsf{Commit}(\mathsf{pp}, (i, \mathsf{I}_i, \mathcal{S}_i, P); r_i)$)
// Compute the $i^{th}$ state encryption key using a PRF on $\mathsf{post}_i.\mathsf{PrevHash}$
$(k_i, \cdot) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{PrevHash})$
if $\mathcal{S}_i = \varepsilon$ then
    // First execution step.  Verify that the counter $i = 0$ and set the state to $\varepsilon$.
    Assert($i = 0$)
    $S_i = \varepsilon$
else
    // Not the first execution step.  Decrypt previous state.
    $(S_i, H_P) \leftarrow \mathsf{Unpad}(\mathsf{Decrypt}(k_i, \mathcal{S}_i))$
    // Ensure that decryption was successful, and check the public output, counter and
       program
    Assert($(S_i, H_P) \neq \perp$)
    Assert($H_P = \mathsf{H}(P \| i \| \mathsf{Pub}_{i-1})$)
// Compute new state encryption key and randomness
$(k_{i+1}, \bar{r}_i) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$
// Run the program and abort if it fails
$(S_{i+1}, \mathsf{Pub}_i, \mathsf{O}_i) \leftarrow P(S_i, \mathsf{I}_i; \bar{r}_i)$
Assert($(S_{i+1}, \mathsf{Pub}_i, \mathsf{O}_i) \neq \perp$)
// Encrypt the resulting state, along with the hash of $i+1$, $P$ and $\mathsf{Pub}_i$
$M \leftarrow \mathsf{Pad}^P(S_{i+1}, \mathsf{H}(P \| i+1 \| \mathsf{Pub}_i))$
$\mathcal{S}_{i+1} \leftarrow \mathsf{Encrypt}(k_{i+1}, M)$
Output $(\mathcal{S}_{i+1}, \mathsf{Pub}_i, \mathsf{O}_i)$
```

Figure 4: ExecuteEnclave algorithm for our main construction. The $\mathsf{Assert}(\cdot)$ primitive terminates execution and outputs $\perp$ if the provided statement evaluates to false.

Deterministic authenticated encryption. We require a symmetric authenticated encryption scheme consisting of the algorithms (Encrypt, Decrypt) where each accepts a key uniformly sampled from $\{0,1\}^\kappa$. It is critical that both algorithms are *deterministic*. This does not require strong assumptions, as we will use Encrypt at most once for any given key; hence standard AE modes can be used if they are configured with a fixed nonce. For simplicity we further assume the specialized algorithm $\mathsf{Encrypt}^n$ will *pad* the plaintext to length n bits prior to encrypting it, and that Decrypt removes this padding.

Pseudorandom Functions. Our construction uses a pseudorandom function family $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \rightarrow \{0,1\}^{2\ell}$ that on input a key k and some string, outputs a 2ℓ-bit pseudorandom string.

Collision-resistant hashing. Our schemes rely on two collision-resistant hash functions $\mathsf{H_L} : \{0,1\}^* \rightarrow \{0,1\}^\ell$ and $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\ell$ where ℓ is polynomial in the scheme's security parameter. For simplicity we do not specify a key for these functions, and we will instead assume that any attack on the scheme implicitly results in the extraction of a hash collision (see *e.g.,* [Rog06]).

3.1 Main Construction

We now present our main construction for a Enclave-Ledger Interaction scheme and address its security. Recall that an ELI consists of the two algorithms with the interface described in Figure 2. We present pseudocode for each of these algorithms in Figures 3 and 4.

Discussion. The scheme we present in this section differs somewhat from the pedagogical scheme we discussed in the introduction. Many of these differences address minor details that affect efficiency or simplify our security analysis: for example we do not encrypt state directly using the fixed key K, but instead derive a unique per-execution key k using a pseudorandom function (PRF). This simplifies our analysis by allowing us to instantiate with a single-message authenticated encryption scheme (*e.g.,* an AE scheme with a hard-coded nonce) without concerns about how to deal with encrypting multiple messages on a single key.

A second modification from our pedagogical construction is that we evaluate this pseudorandom function on the hash of the *structure* returned by the ledger. The intuition is that this data structure enforces a hash chain over all previous transactions; as a result this ensures that all random coins and keys are themselves a function of the *full execution history of the program.* This ensures that an attacker – even a powerful one that can forge some ledger outputs – cannot use the state resulting from those forgeries to continue normal execution via the real ledger, since the execution history on the real ledger will not contain these forgeries.

We remark again that this more powerful ledger abstraction does not truly represent a stronger assumption when compared to our pedagogical construction, since the more powerful ledger can be "simulated" by enclave itself, provided the enclave has access to the full contents of a simple ledger.

Security We now present our main security theorem.

Theorem 3.1 Assuming a secure commitment scheme, a secure authenticated encryption scheme (in the sense of [Rog02]); that H and $\mathsf{H_L}$ are collision resistant; PRF is pseudorandom; and that ledger authentication tags are unforgeable,[13] then the scheme $\Pi = (\mathsf{Setup}, \mathsf{ExecuteEnclave}, \mathsf{ExecuteApplication})$ presented in Figures 3 and 4 satisfies Definition A.1.

We present a proof of Theorem 3.1 in Appendix B.

4 Applications

We now describe several applications that use Enclave-Ledger Interaction and present the relevant implementations for each. Each application employs the main construction we presented in §3 to implement a specific functionality. Except where explicitly noted, these applications are implemented as *many-time execution* programs: this means the host can re-launch the same program P many times, but each execution thread is independent and threads do not share state.

4.1 Private Smart Contracts

Smart contract systems comprise a network of volunteer nodes that work together to execute multi-step interactive programs called *contracts.* These systems, which are exemplified by Ethereum and the Hyperledger platforms [eth, Hyp17] maintain a shared ledger that records both the previous and updated state of the contract following each execution of a contract program. These platforms are designed for flexibility: they are capable of executing many different contracts on a single network.

Smart contract systems come in two varieties: *public* contract networks (exemplified by Ethereum [eth]), where all state and program code is known to the world; and *private* contract systems where some portion of this data is held secret. In both settings the computation (and verification) is conducted by a set of nodes who are not assumed to always be trustworthy. In the public setting (*e.g.,* [eth]) a single node performs each contract execution, and the remaining nodes simply verify the (deterministic) output of this calculation. This

[13] We discuss this property in Appendix 2.2.

approach does not work in the private setting, where some of the program inputs are unknown to the full network.

Platforms such as Hyperledger Sawtooth [Int] have sought to address this concern by employing trusted execution technology [Hyp17]. In these systems, contract code executes within a trusted enclave on a single node, and the TEE system generate public *attestation* signatures proving the correctness of the resulting output. The network then verifies the attestation to ensure that the execution was correct. A challenge in these systems is to ensure that a smart contract remains *synchronized*, despite the fact that execution migrates from one host to another between steps. A secondary challenge is to ensure that the enclave only executes contract code on valid inputs from the ledger, and cannot be forced to run on arbitrary input or perform additional steps by a malicious host.

Bowman *et al.* of Intel corporation [Bow17] independently proposed a solution reminiscent of an ELI for securing contracts in this setting.[14] (Figure 1 presents an illustration of this model.) This setting is also a natural solution for our ELI system, given that the contract system – with many distinct enclave copies – can be viewed as merely being a special case of our main construction.

To instantiate an ELI in this setting, we require the contract author to pre-position a key K within each enclave.[15] Encrypted state outputs can now be written to the ledger as a means to distribute them. Given these modifications, each enclave can simply read the current state from the ledger in order to obtain the most recent encrypted state and commitment to the next contract input (which may be transmitted by users to the network).[16] Other enclaves can *verify* the correctness of the resulting output state by either (1) verifying an attestation signature, or (2) deterministically re-computing the new state and comparing it to the encrypted state on the ledger.

4.2 Logging and Reporting

Several cryptographic access control systems require participants to actively log file access patterns to a remote and immutable network location [Fou]. A popular approach to solving this problem in cryptographic access control systems, leveraged by systems like Hadoop [Fou], is to assign a unique decryption key to each file and to require that clients individually request each key from an online server, which in turn logs each request. This approach requires a trusted online server that holds decryption keys and cannot be implemented using a public ledger.

In place of a trusted server, we propose to use ELI to implement mandatory logging for protected files. In this application, a local enclave is initialized (in the first step of a program) and stores (or generates) a master key for some collection of files, *e.g.,* a set of files stored on a device.[17] The enclave then employs the *public output* field of the ELI scheme to ensure that prior to each file access the user must post a statement signaling that the file is to be accessed.[18] The logging program is presented as Algorithm 4 and consists of three phases. When the program is launched, the enclave generates a keypair for a public-key encryption scheme $(\mathsf{PKKeyGen}, \mathsf{PKEnc}, \mathsf{PKDec})$ and outputs the public key.[19] Next the user provides a filename they wish to decrypt, and the program encrypts this filename using a hard-coded public key for an *auditor*. When the user posts this key to the ledger, the program decrypts the given file.

[14]At present the Bowman *et al.* solution is unpublished, but Bowman informs the authors that Intel is developing it internally as a product. Intel has not performed formal analysis of their solution.

[15]This can be accomplished using a broadcast encryption scheme or peer-to-peer key sharing mechanism.

[16]We assume that the ledger is authenticated using signatures. Systems such as Hyperledger propose to use TEE enclaves to construct the ledger as well as execute contracts; in these systems the ledger blocks are authenticated using digital signatures that can be publicly verified.

[17]If the Enclave is implemented using cryptographic techniques such as FWE, a unique Enclave can be shipped along with the files themselves. If the user employs a hardware token, the necessary key material can be delivered to the user's Enclave when the files are created or provisioned onto the user's device.

[18]To provide confidentiality of file accesses, the enclave may encrypt the log entry under the public key of some auditing party.

[19]Here we require the encryption scheme to be CCA-secure.

Algorithm 4: File Access Logging $P_{logging}$

```
Data: Input: $\mathsf{I}_i$ , $\mathsf{S}_i$; Constants: $pk_{\mathsf{auditor}}$
Parse $(\mathsf{phase}, sk, CT, \mathsf{filename}) \leftarrow \mathsf{S}_i$
if $\mathsf{S}_i = \varepsilon$ then // Generate master keypair
    $(pk, sk) \leftarrow \mathsf{PKKeyGen}(1^\lambda)$
    $\mathsf{S}_{i+1} = (\mathsf{PUBLISH}, sk, \cdot, \cdot)$
    $(\mathsf{Pub}_{i+1}, \mathsf{O}_{i+1}) \leftarrow (\varepsilon, pk)$
else if $\mathsf{phase} = \mathsf{PUBLISH}$ then // Send filename
    Parse $\mathsf{filename} \leftarrow \mathsf{I}_i$
    $CT_{\mathsf{auditor}} \leftarrow \mathsf{PKEnc}(pk_{\mathsf{auditor}}, \mathsf{filename})$
    $\mathsf{S}_{i+1} \leftarrow (\mathsf{DECRYPT}, sk, CT, \mathsf{I}_i)$
    $(\mathsf{Pub}_{i+1}, \mathsf{O}_{i+1}) \leftarrow (CT, \mathsf{post})$
else // Decrypt a given file
    Parse $C \leftarrow \mathsf{I}_i$
    $(\mathsf{filename}', M) \leftarrow \mathsf{PKDec}(sk, C)$
    if $\mathsf{filename} = \mathsf{filename}'$ then
        $\mathsf{S}_{i+1} = (\mathsf{PUBLISH}, sk, \cdot, \cdot)$
        $(\mathsf{Pub}_{i+1} \| \mathsf{O}_{i+1}) = (\varepsilon \| M)$
    else
        Abort and output $\perp$.
output $(\mathsf{S}_{i+1}, \mathsf{Pub}_{i+1}, \mathsf{O}_{i+1})$
```

4.3 Limited-attempt Password Guessing

Device manufacturers have widely deployed end-to-end file encryption for devices such as mobile phones and cloud backup data [App16, And]. These systems require users to manage their own secrets rather than trusting them to the manufacturer.

Encryption requires high-entropy cryptographic keys, but users are prone to lose or forget high-entropy passwords. To address this dilemma, manufacturers are turning to *trusted hardware*, including on-device cryptographic co-processors [App16]. trusted enclaves [ARM17], and cloud-based HSMs [Las17, Krs16] for backup data. A user authenticates with a relatively weak passcode such as a PIN and the hardware will release a strong encryption key. To prevent passcode guessing attacks, this *stateful* hardware must throttle or limit the number of login attempts.[20]

Enclave-Ledger Interaction provides an alternative mechanism for limiting the number of guessing attempts on password-based encryption systems. A manufacterer can employ an inexpensive stateless hardware token to host a simple enclave, with an internal (possible hard-wired) secret key K. In the initial step, the enclave takes in a password uses the random coins to produce a master encryption key k_{enc} that it outputs to the user. The Enclave is constructed to release k_{enc} only when it is given the proper passcode *and* the step counter is below some limit. Note that if the host restarts the execution, this simply re-runs the setup step which will generate a new key unrelated to the original. Rate limiting can be accomplished if the ledger has some approximation of a clock, like number of blocks between login attempts in Bitcoin. In practice the decryption process in such a system can be fairly time consuming if the ledger has significant lag. This system may be useful for low frequency applications such as recovering encrypted backups or emergency password recovery.

[20]This approach led to the famous showdown between Apple and the FBI in the Spring of 2016. The device in question used a 4-character PIN, and was defeated in a laboratory using a state rewinding attack, and in practice using an estimated \$1 million software vulnerability [Pal15, Wea15].

Algorithm 5: Ransomware $P_{ransomware}$

```
Data: Input: $\mathsf{I}_i$ , $\mathsf{S}_i$; Randomness $r_i$;
Parse $(K, \mathsf{R}, pk) \leftarrow \mathsf{S}_i$
if $\mathsf{S}_i = \varepsilon$ then // Generate Key, Set Ransom
    Parse $(\mathsf{R}, pk) \leftarrow \mathsf{I}_i$
    $K \leftarrow \mathsf{KDF}(r_i)$
    output $\mathsf{S}_{i+1} \leftarrow (K, \mathsf{R}, pk)$
else // Release Key on Payment
    Parse $(t, \sigma) \leftarrow \mathsf{I}_i$
    if $(\mathsf{BlockchainVerify}(t, \sigma) = 1)$ then
        if $(t.amount > \mathsf{R}$ and $t.target = pk)$ then
            output $\mathsf{O}_i = K$
    output $\mathsf{O}_i = \perp$
```

4.4 Paid Decryption and Ransomware

ELI can also be used to condition program execution on *payments* made on an appropriate payment ledger such as Bitcoin or Ethereum. Because in these systems payment transactions are essentially just transactions written to a public ledger, the program P can take as input a public payment transaction and condition program execution on existence of this transaction. This feature enables pay-per-use software with no central payment server.

Not all of the applications of this primitive are constructive. The ability to condition software execution on payments may enable new types of destructive application such as ransomware [Zet16]. In current ransomware, the centralized system that deliver keys represent a weak point in the ransomware ecosystem. Those systems exposes ransomware operators to tracing [Tec16]. As a result, some operators have fled without delivering key material, as in the famous WannaCry outbreak [Kan17].

In the remainder of this section we consider a potential *destructive* application of the ELI paradigm: the development of *autonomous* ransomware that guarantees decryption without the need for online C&C. We refer to this malware as autonomous because once an infection has occurred it requires no further interaction with the malware operators, who can simply collect payments issued to a Bitcoin (or other cryptocurrency) address.

In this application, the malware portion of the ransomware samples an encryption key $K \in \{0,1\}^\ell$ and installs this value along with the attackers public address within a Enclave. The Enclave will only release this encryption key if it is fed a validating blockchain fragment containing a transaction paying sufficient currency to the attacker's address. Algorithm 5 presents a simple example of the functionality.

We note that the Enclave may be implemented using trusted execution technology that is becoming available in commercial devices, *e.g.,* an Intel SGX enclave, or an ARM TrustZone trustlet. Thus, autonomous ransomware should be considered a threat today – and should be considered in the threat modeling of trusted execution systems. Even if the methods employed for securing these trusted execution technologies are robust, autonomous ransomware can be realized with software-only cryptographic obfuscation techniques, if such technology becomes practical [LMA$^+$16].

This application can be extended by allowing a ransomware instance to prove to a skeptical victim that it contains the true decryption key without allowing the victim to regain all their files. The victim and the ransomware can together select a random file on the disk to decrypt, showing the proper key is embedded. Additionally, the number of such files that can be decrypted can be limited using similar methodology as in Section 4.2.

4.5 Fair Encryption and MPC

In collaboration with the current authors [CGJ$^+$17], Choudhuri *et al.* explore a similar model to ELI to acheive fair multiparty computation without requiring an honest majority of parties. Fundemental to acheiving their goal is constructing an encryption in which a verifiable public deloration acts as a decryption key. They accomplish this goal in a theoretical result using Extractable Witness Encryption and reduce it to prace using Intel SGX. This work represents a single specific application of the significantly more powerful Enclave-Ledger Interaction.

We generalize this notion as *Fair Encryption.* Intuitively, this primitive ensures that if one authorized decryptor can decrypt a ciphertext, then *all* authorized decryptors may do so as well. We accomplish this by constructing an enclave that will decrypt a CCA2-secure ciphertext if and only if a valid ciphertext has been placed on the public consensus network transcript by a Ledger. The details of constructing such an encryption scheme can be found in [CGJ$^+$17], but the applications of such a primative apply well beyond multi-party computation. For instance consider a program that requires a broadcast channel. Because the broadcast channel is mediated by a malicious user, an unflattering message may be blocked. A ciphertext that can only be decrypted when it becomes public circumvents the tampering of the user.

5 Realizing the Enclave and Ledger

In this section we turn to the matter of instantiating the enclave and ledger components of a real ELI system.

5.1 Realizing the Enclave

Trusted cryptographic co-processors. The simplest approach to implement the enclave is using a secure hardware or trusted execution environment such as Intel's SGX [sgx] or ARM Trustzone [ARM17]. When implemented using these platforms, our techniques can be used immediately for applications such as logging, fair encryption and ransomware.

While these environments provide some degree of hardware-supported immutable statekeeping, this support is surprisingly limited. For example, Intel SGX-enabled processors provide approximately 200 monotonic counters to be shared across all enclaves. On shared systems these counters could be maliciously reserved by enclaves such that they are no longer available to new software. Finally, these counters do not operate across enclaves operating on different machines, as in the smart contract setting.

Many simpler computing devices such as smart cards lack any secure means of keeping state. In our model, even extremely lightweight ASICs and FPGAs could be used to implement the enclave for stateful applications using our ELI constructions. Along these lines, Nayak *et al.* [NFR$^+$17] recently showed how to build trusted non-interactive Turing Machines from minimal *stateless* trusted hardware. Such techniques open the way for the construction of arbitrary enclave functionalities on relatively inexpensive hardware.

Software-Only Options. A natural software-only equivalent of the enclave is to use pure-software techniques such as virtualization, or cryptographic program obfuscation [BGI$^+$01]. While software techniques may be capable of hiding secrets from an adversarial user during execution, interactive multi-step obfuscated functionalities are *implicitly* vulnerable to being run on old state. Unfortunately, there are many negative results in the area of program obfuscation [BGI$^+$01], and current primitives are not yet practical enough for real-world use [LMA$^+$16]. However, for specific functionalities this option may be feasible: for example, Choudhuri *et al.* [CGJ$^+$17] and Jager *et al.* [Jag15] describe protocols based on the related Witness Encryption primitive. New developments in this area could make practical constructions feasible in the next several years.

5.2 Realizing the Ledger

There are many different systems that may be used to instantiate the ledger. In principle, any stateful centralized server capable of producing SUF-CMA signatures can be used for this purpose. There are a number properties we require of our ledgers: (1) the unforgeability of the authentication tags, (2) public verifiability of authenticators, and (3) in our more efficient instantiations, the ability to compute and return transaction hashes. We describe four potential realizations in detail: unstructured public ledgers such as Certificate Transparency [cer18], proof of work blockchains like Bitcoin, smart contract systems like Ethereum, and private blockchains.

Remark: Recall that in our setting, the Post algorithm is run by the host, and the Verify algorithm is executed by the Enclave. We assume that an honest host can keep state locally, although the security of an ELI scheme does not assume an honest host. In the presentation below, we modify the Ledger Post interface by adding an additional argument aux. This argument contains additional state stored by the host. We note that this does not change the functioning of the algorithm or the security of the system, but does require the host to record and store this data.

Certificate Transparency A number of browsers have begun to mandate *Certificate Transparency* (CT) proofs for TLS certificates [cer18]. In these systems, every CA-issued certificate is included in a public log, which is published and maintained by a central authority such as Google. Every certificate in the log is included as a leaf in a Merkle tree, and the signed root and associated membership proofs are distributed by the log maintainer.

Provided that the log maintainer is trustworthy, this system forms a public append-only ledger with strong cryptographic security. The inclusion of a certificate can be verified by any party who has the maintainer's public key, while the tree location can be viewed as a unique identifier of the posted certificate. Because many certificate authorities support CT, the ability to programmatically submit certificate signing requests allows us to use CT as a log for any arbitrary data that can be incorporated into an X.509 certificate.[21] In our presentation we implicitly assume that the Enclave can verify CT inclusion proofs from a specific log *i.e.,* that it has been provisioned with a copy of the log maintainer's public verification key.

A limitation of the CT realization is that, to implement our Ledger functionality of §2.2, we require a way to ensure that the $\mathsf{PrevHash}$ field of each record truly does identify the previous entry in the log. Unfortunately, the current instantiation of CT does not guarantee this; instead, the enclave must read *the entire certificate log* to verify that no interceding entries exist. This makes CT less bandwidth-efficient than the other realizations.

We can use Certificate Transparency as the ledger for an ELI construction as follows. We modify the Post algorithm to take in an additional argument aux that contains metadata for use in posting. This data is provided by the host.

- $\mathsf{CT.Post}(\mathsf{Data}, \mathsf{CID}, \mathsf{aux}) \rightarrow (\mathsf{post}, \sigma, \mathsf{aux}')$.

  1. Parse CID as vk, where vk is the public key of an $\mathsf{SUF\text{-}CMA}$ signature scheme.
  2. Parse aux as $(sk, \mathsf{index_{prev}})$ where sk is the corresponding secret key for vk, and $\mathsf{index_{prev}}$ is the index into the log of the most recent certificate posted under CID. If no previous index exists, set $\mathsf{index_{prev}} \leftarrow \perp$.
  3. Query the CT log to obtain the certificate $\mathsf{Cert_{prev}}$ located at index $\mathsf{index_{prev}}$.
  4. Generate a new X.509 certificate Cert embedding the public verification key vk, along with the following custom X.509 fields:[22]
     (a) $\mathsf{Cert.Data} \leftarrow \mathsf{Data}$.
     (b) $\mathsf{Cert.index_{prev}} \leftarrow \mathsf{index_{prev}}$.
     (c) If $\mathsf{index_{prev}} \neq \perp$ then set $\mathsf{Cert.PrevHash} \leftarrow \mathsf{H_L}(\mathsf{Cert_{prev}.Data} \| \mathsf{Cert_{prev}.PrevHash})$.
     (d) If $\mathsf{index_{prev}} = \perp$ then set $\mathsf{Cert.PrevHash} \leftarrow \textbf{Root} : vk$.
  5. Sign the certificate using sk.
  6. Post Cert to the CT ledger. Obtain the resulting log index index.

[21] This can be done at no cost and with no manual interaction using *e.g.,* the free LetsEncrypt CA.

[22] In practice this data can be encoded in any fashion within the certificate, including as strings within standard fields.

7. Set $\mathsf{post.Data} \leftarrow \mathsf{Cert.Data}$.
8. Set $\mathsf{post.PrevHash} \leftarrow \mathsf{Cert.PrevHash}$.
9. Set $\mathsf{post.Hash} \leftarrow \mathsf{H_L}(\mathsf{post.Data} \| \mathsf{post.PrevHash})$.
10. Obtain all certificates $(\mathsf{Cert}_1, \ldots, \mathsf{Cert}_\ell)$ representing the entire CT log, along with the corresponding inclusion proofs $(\bar{\sigma}_1, \ldots, \bar{\sigma}_\ell)$. Set $\sigma \leftarrow (vk, \mathsf{index}, \mathsf{Cert}_1, \ldots, \mathsf{Cert}_\ell, \bar{\sigma}_1, \ldots, \bar{\sigma}_\ell)$. Set $\mathsf{aux}' = (sk, \mathsf{index})$.

Output (post, σ).

- $\mathsf{CT.Verify}(\mathsf{post}, \sigma) \rightarrow \{0, 1\}$.

1. Parse σ as $(vk, \mathsf{index}, \mathsf{Cert}_1, \ldots, \mathsf{Cert}_\ell, \bar{\sigma}_1, \ldots, \bar{\sigma}_\ell)$.
2. Verify each inclusion proof $\bar{\sigma}_1, \ldots, \bar{\sigma}_\ell$ against the corresponding certificate in $\mathsf{Cert}_1, \ldots, \mathsf{Cert}_\ell$. Abort and output 0 if any certificate fails to verify, or if the list contains any gaps (*i.e.,* missing certificates).
3. Scan the list $\mathsf{Cert}_1, \ldots, \mathsf{Cert}_\ell$ and verify that there is exactly one index i where $\mathsf{Cert}_i.\mathsf{PrevHash} = \textbf{Root} : vk$. If no such certificate exists, or if *multiple* matching certificates are found, abort and output 0.
4. Beginning with the index i discovered above, repeat the following steps:
   (a) Verify that Cert_i embeds the public key vk, and that it has a valid signature under vk. If either condition is false, abort and output 0.
   (b) Beginning with Cert_i, scan forward to find every index $j > i$ where (1) $\mathsf{Cert}_j.\mathsf{index}_{\mathsf{prev}} = i$ and (2) Cert_j embeds vk and (3) the certificate contains a signature that verifies using vk. If two such certificates are found, abort and output 0.
   (c) If no satisfying j is found, and $i \neq \mathsf{index}$ then abort and output 0.
   (d) If no satisfying j is found, and $i = \mathsf{index}$ then exit this loop and continue with the next step.
   (e) Verify that $\mathsf{Cert}_j.\mathsf{PrevHash} = \mathsf{H_L}(\mathsf{Cert}_i.\mathsf{Data} \| \mathsf{Cert}_i.\mathsf{PrevHash})$. If this check does not succeed, abort and output 0.
   (f) Set $i \leftarrow j$ and go back to step 4a.
5. Verify that $\mathsf{Cert}_{\mathsf{index}}.\mathsf{Data} = \mathsf{post.Data}$.
6. Verify that $\mathsf{Cert}_{\mathsf{index}}.\mathsf{PrevHash} = \mathsf{post.PrevHash}$.
7. Verify that $\mathsf{post.Hash} = \mathsf{H_L}(\mathsf{post.Data} \| \mathsf{post.PrevHash})$.
8. If all the above checks pass, output 1.

Discussion. The construction above is similar to the pedagogical description given in the Introduction, with some changes required to satisfy the Ledger interface given in 2.2. The main difference in this construction is that the host signs each certificate with a signing key it generates, to ensure that *only* the host can submit certificates as part of a chain. To facilitate this, the "chain identifier" CID is a set to be the verification key for an SUF-CMA signature scheme and generated by the host. The enclave simply validates locally that only one matching chain exists, that the chain does not contain forks, and that all hashes are correctly computed; it can do this because it receives the full contents of the CT ledger.

Bitcoin and Proof-of-work Blockchains Public blockchains, embodied most prominently by Bitcoin, are designed to facilitate distributed consensus as to the contents of a ledger. In these systems, new blocks of transactions are added to the ledger each time a participant solves a costly *proof of work* (PoW), which typically involves solving a hash puzzle over the block contents. These PoW solutions are publicly verifiable, and can be used as a form of "economic" authentication tag over the block contents: that is, while these tags can be forged, the financial cost of doing so is extremely high. Moreover, because blocks are computed in sequence, a sub-chain of n blocks (which we refer to as a "fragment") will include n chained proofs-of-work, resulting in a linear increase in the cost of forging the first block in the fragment.

The remaining properties of our ledger are provided as follows: in Bitcoin, transactions are already uniquely identified by their hash, and each transaction (by consensus rules) must identify a previous transaction hashes as an *input*. Similarly, due to double spending protections in the consensus rules, there cannot be two transactions that share a previous input. Finally, we can encode arbitrary data into the transaction using the `OP_RETURN` script [opr18]. In this description, we will assume that the Bitcoin transaction

We define the interface as follows:

- $\mathsf{Bitcoin.Post}(\mathsf{Data}, \mathsf{CID}, \mathsf{aux}) \to (\mathsf{post}, \sigma, \mathsf{aux}')$.

  1. Parse CID as the transaction identifier of a Bitcoin transaction. Parse aux as BTX' and a bitcoin secret key, where BTX' is the most recent in the chain of transactions rooted at transaction ID CID.
  2. Construct a new Bitcoin transaction BTX such that:
     (a) $\mathsf{BTX.vout}[0]$ spends most of the output to an address owned by the user.
     (b) $\mathsf{BTX.vout}[1] \leftarrow (\texttt{OP\_RETURN}\ \mathsf{Data})$.
     (c) $\mathsf{BTX.vin}[0]$ spends $\mathsf{BTX}'.\mathsf{vout}[0]$.
     (d) The transaction is correctly signed with the user's secret key.
  3. Now send BTX to the Bitcoin network and receive a block B containing BTX as well as b subsequent blocks.
  4. Let post represent the appropriate subset of BTX.
  5. Output BTX and the secret key.

  The ledger sets σ to be the block B containing BTX and the subsequent b blocks referencing B. Implicitly we treat B as the transaction post, where $\mathsf{post.Hash}$ is the SHA256 transaction hash of BTX, $\mathsf{post.PrevHash}$ is the SHA256 hash of BTX', and $\mathsf{post.Data}$ is the contents following the `OP_RETURN` field in $\mathsf{BTX.vout}[1]$.

  Note that the consensus rules will not allow the network to accept the transaction BTX if the preceding transaction BTX' is already spent.

- $\mathsf{Bitcoin.Verify}(\mathsf{post}, \sigma) \to \{0, 1\}$.

  1. Verify that post contains the appropriate fields of BTX.
  2. Parse post and σ as BTX, B and b subsequent confirmation blocks.
  3. Verify that the transaction BTX is well formatted as above and contained within B.
  4. Verify that the proof of work on the block B is valid and verify that the proof of work on each of the b confirmation blocks is valid.[23]. If not, return 0. Otherwise return 1.

Analyzing the cost of forging blockchain fragments. Proof-of-work blockchains do not provide a cryptographic guarantee of unforgeability. To provide some understanding of the cost of forging in these systems, we can examine the economics of real proof-of-work blockchains. We propose argue that the cost of forging an authenticator can can be determined based on from block reward offered by a proof-of-work cryptocurrency, assuming that the market is liquid and reasonably efficient.

In currencies such as Bitcoin, the reward for producing a valid proof-of-work block is denominated in the blockchain currency, which has a floating value with respect to currencies such as the dollar. Critically, because each instance of a PoW puzzle in the real blockchain is based on the preceding block, an adversarial miner must choose at mining time if they want to mine on the blockchain or attempt to forge a block for use in the ELI scheme; their work cannot do double duty. Thus we can calculate the opportunity cost of forgoing normal mining in order to attack an ELI system: the real cost of forging a block is at least the value of a block reward. Similarly, the cost of forging a blockchain fragments of length n is at least n times the block reward. At present, the cost of forging a fragment of length 7 would be 87.5 BTC.

Remark. This simple analysis ignores that a single blockchain fragment may be used by multiple instances of a given enclave. This admits the possibility that an attacker with significant capital might amortize this

[23]We abstract away the implementation detail of block difficulty. The validation algorithm can require a minimum difficulty.

cost by spreading it across many instances. Indeed, if amortized over a sufficient number of forged ledger posts, this fixed cost could be reduced. For scenarios where we expect sufficient instances for this attack to be practical, it is necessary to rate limit the number of ledger posts included in a given block that the enclave will accept results from.

Ethereum and Smart Contract Systems. A very natural realization of our ledger system is a smart contract systems such as Ethereum [Nak08, eth]. Smart contract systems enable distributed public computation on the blockchain. Typically, a program is posted to some specific address on the blockchain. When a user submits a transaction to the associated address, the code is executed and appropriate state is updated. As noted previously, our system allows for smart contracts with private data, which is impossible on current implementations of smart contract system. We define the interface of a smart contract ledger as follows:

- $\mathsf{Ethereum.Post}(\mathsf{Data}, \mathsf{CID}) \rightarrow (\mathsf{post}, \sigma)$.

  1. Parse CID as `contract_address`, a contract address in the Ethereum system.
  2. Initiate a contract call to the contract `contract_address` on argument Data.
  3. The contract at `contract_address` must maintain a state variable $\mathsf{PrevHash}$ that contains the hash of the last message it received (or **Root** : CID for the first call).
  4. The contract embeds Data into its output along with the previous message hash.
  5. The contract updates $\mathsf{PrevHash} \leftarrow \mathsf{H_L}(\mathsf{PrevHash} \| \mathsf{Data})$

  The ledger computes an authentication tag σ the contract's response using the proof-of-work consensus mechanism and confirmation blocks. The ledger returns $(\mathsf{post}_, \sigma)$.

- $\mathsf{Ethereum.Verify}(\mathsf{post}, \sigma) \rightarrow \{0, 1\}$.

  1. Verify the proof-of-work and confirmation blocks, and ensure that the contract contains the correct program. Return 1 if and only if all the verification passes.

Remark: While our construction, as mentioned in Section 2, is sufficient for the creation of multi-use programs, the protocol in Section 3 is unable to realize the stronger notion of one-time programs. Intuitively this is because a host can invoke a first step - that is a step without any previous ciphertext - on a fresh CID and the enclave will be unable to detect this action. Smart contract systems allow a clean solution to this problem. We add a single check to the enclave execution algorithm: the enclave checks that the contract address is output of the PRF evaluated on some special value. If the enclave will only accept transaction to a particular address, the host is unable to restart computation on a different CID, so the program can only be run once.

Private Blockchains. Many recent systems such as Hyperledger [Hyp17] implement private smart contracts by constructing a shared blockchain among a set of dedicated nodes. In some instantiations, the parties forgo the use of proof-of-work in favor of using digital signatures and trusted hardware to identify the party who writes the next block [Int]. Private blockchains represent a compromise between centralized systems such as CT and proof-of-work blockchains. They are able to use digital signatures to produce ledger authentication tags so the security is not economic in nature. Moreover, the ledger can be constructed to provide efficient rules for ledger state updates, which enables an efficient realization of our model of §2.2.

6 Prototype Implementation

To validate our approach we implemented our ELI construction using Intel SGX to implement the enclave, and the Bitcoin blockchain to implement the ledger.[24] For details on the SGX computation model we refer the reader to [sgx, MAB+13, JSR+16, AGJS13, Rao16, Int16, B16].

[24]The source code can be found at `https://github.com/JHU-ARC/state_for_the_stateless/`

In our system, the host application communicates with a local Bitcoin node via RPC to receive blockchain (ledger) fragments for delivery to the enclave, and to sends transaction when requested by the enclave. To verify blockchain fragments and verify proof-of-work tags – which we use as our authentication tags for the ledger – the enclave requires an independent (partial) Bitcoin implementation. We based this on the C++ SGX-Bitcoin implementation in the Obscuro project [TLK+17].

We require a language for describing the program P. For this purpose we embedded a lightweight Javascript engine called Duktape [duk18] into our enclave. While this is prototype code and not designed for production security, the use of an interpreted language protects the integrity of the SGX Enclave. Executing arbitrary C code inside an enclave might allow for secret exfiltration. Hypothetically, Javascript code can executed safely inside of a sandbox.[25]

At startup, the host application loads the Javascript program from a file, initializes the protocol values as in Figures 3 and 4 and launches the SGX enclave. At first initialization the enclave generates a random long term master key K, which can be sealed to the processor using SGX's data sealing interface, protecting the key from power fluctuations. In each iteration of the protocol, the untrusted application code prompts the user for the next desired input. It then generates a transaction T using `bitcoin-tx` RPC. The first "input" $T.\mathsf{vin}[0]$ is set to be an unspent transaction in the local wallet. The first "output" $T.\mathsf{vout}[0]$ spends the majority of the input transaction to a new address belonging to the local wallet. The second output $T.\mathsf{vout}[1]$ embeds $\mathsf{SHA256}(i \| \mathsf{I}_i \| \mathcal{S}_i \| P \| \mathsf{CID} \| r_i)$ in an `OP_RETURN` script. The third output $T.\mathsf{vout}[2]$ embeds the public output Pub emitted by the previous step. This transaction is signed by a secret key in the local wallet and submitted for confirmation.

The host application now monitors the blockchain until T has been confirmed by 6 blocks.[26] The host then formats the input structures:

- $\mathsf{post}_i.\mathsf{Data} \leftarrow T.\mathsf{vout}$
- $\mathsf{post}_i.\mathsf{PrevHash} \leftarrow T.\mathsf{vin}[0].Hash$
- $\mathsf{post}_i.\mathsf{CID} \leftarrow$ chain of transactions from T back to the transaction with hash $\mathsf{post}_0.\mathsf{PrevHash}$
- $\mathsf{post}_i.\mathsf{Hash} \leftarrow T.Hash$
- $\sigma_i \leftarrow$ 6 blocks confirming T

The host then submits $(\mathsf{post}_i, \sigma_i)$ to the enclave which then performs the following checks: (1) verifies that σ_i is valid and has sufficiently high block difficulty (2) the blocks in σ_i are consecutive (3) $T.\mathsf{vout}[0], T.\mathsf{vout}[1]$ embed the correct data and (4) the transactions in $\mathsf{post}_i.\mathsf{CID}$ are well formatted.

If $i = 0$ and there is no input state, the enclave generates a zero initial state. Otherwise it generates the decryption key as described in the protocol using $\mathsf{C\text{-}MAC}$ to implement the PRF. The state along with the inputs and random coins are passed to the Javascript interpreter. All hashes computed in the enclave are computed using $\mathsf{SHA256}$. One note is that instead of hashing all of CID into the ciphertext, we include only $\mathsf{post}_0.\mathsf{PrevHash}$, which keeps CID constant throughout the rounds.

Measurements To avoid spending real money on the Bitcoin main network, we tested our implementation on a private regression $\mathsf{regtest}$. This also allows us to control the rate at which blocks are mined. The most time-consuming portion of an implementation using the $\mathsf{mainnet}$ or $\mathsf{testnet}$ is waiting for blocks to be confirmed; blocks on the main bitcoin network take an average of 10 minutes to be mined, or an average of 70 minutes to mine a block and its 6 confirmation blocks. If an application requires faster execution, alternative blockchains can be used, such as Litecoin (2.5 minutes per block) or Ethereum (approximately 10-19 seconds).

[25]A hardened implementation might separate the ELI protocol code (and key k) from the Javascript interpreter by placing these into two separate enclaves and using Intel's intra-enclave communication features.

[26]In general, six blocks is considered sufficiently safe for normal Bitcoin payment operations; however the number of confirmations blocks can be tweaked as an implementation parameter.

Computation Section	Running Time	Percentage
Bitcoin Operations	**7764 μs**	**100%**
Proof Preparation	7094 μs	92.8%
Proof Verification	550 μs	7.2%
Protocol Operations	**2006 μs**	**100%**
Ciphertext Decryption	4 μs	0.2%
Javascript Invocation	1920 μs	95.7%
Ciphertext Encryption	82 μs	4.0%
SGX Overhead	**1153348 μs**	**100%**
Enclave Initialization	1153308 μs	100.0%
Ecall Entry and Exit	40 μs	0.0%

Figure 5: Measured computation overhead for different elements of our ELI experiment using a simple string concatenation program P. Because SGX does not support internal time calls, these times were measured by the application code. The table above shows averaged results over 100 runs on the Bitcoin regtest network, and isolates just the computing time of each step individually.

Our experiments used a trivial string concatenation program P. For our experiments we measured three specific operations: (1) the execution time of the Bitcoin operations (on the host, enclave and regtest network, (2) ELI protocol execution time, (3) the time overhead imposed by Intel SGX operations. Figure 5 shows the running times of these parts of our implementation. It is worth noting that SGX does not provide access to a time interface, and there is no way for an SGX enclave to get trustworthy time from the operating system. The times in Figure 5 were measured from the application code.

Discussion. Note that initializing an SGX enclave is a one-time cost that must be paid when the enclave is first loaded into memory. It is a comparatively expensive operation because the SGX driver must verify the code integrity and perform other bookkeeping operations. An additional computationally expensive operation is obtaining the proof-of-publication to be delivered to the enclave. This process relies on `bitcoin-cli` to retrieve the proper blocks, which can be slow depending on the status of the `bitcoind` daemon. We note that these tests were run using the regression blockchain regtest, and retrieving blocks from testnet or mainnet may produce different results.

7 Conclusion

In this work we considered the problem of constructing secure stateful computation from limited computing devices. This work leaves several open questions. First, while we discussed the possibility of using cryptographic obfuscation schemes to construct the enclave, we did not evaluate the specific assumptions and capabilities of such a system. Additionally, there may be other capabilities that the enclave-ledger combination can provide that are not realized by this work. Finally, while we discussed a number of applications of the ELI primitive, we believe that there may be many other uses for these systems.

References

[AGJS13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[And] Android Project. File-based encryption for android. Available at `https://source.android.com/security/encryption/file-based`, 2017.

[App16] Apple Computer. iOS Security: iOS 9.3 or later. Available at https://www.apple.com/business/docs/iOS_Security_Guide.pdf, May 2016.

[App17] Apple Computer. Answers to your questions about Apple and security. Available at http://www.apple.com/customer-letter/answers/, 2017.

[ARM17] ARM Consortium. ARM Trustzone. Available at https://www.arm.com/products/security-on-arm/trustzone, 2017.

[B16] Alexander B. Introduction to Intel SGX Sealing. Available at https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing, 2016.

[BFGM01] Mihir Bellare, Marc Fischlin, Shafi Goldwasser, and Silvio Micali. Identification protocols secure against reset attacks. In Birgit Pfitzmann, editor, *EUROCRYPT '01*, pages 495–511, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. Cryptology ePrint Archive, Report 2001/069, 2001. http://eprint.iacr.org/2001/069.

[Bon12] Joseph Bonneau. The science of guessing: Analyzing an anonymized corpus of 70 million passwords. In *IEEE S&P (Oakland) '12*, SP '12, pages 538–552, Washington, DC, USA, 2012. IEEE Computer Society.

[Bow17] Mick Bowman. Personal communication, 2017.

[cer18] Certificate transparency. Available at https://www.certificate-transparency.org, 2018.

[CGJ$^+$17] Arka Rai Choudhuri, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers. Fairness in an unfair world: Fair multiparty computation from public bulletin boards. In *CCS '17*, 2017. https://eprint.iacr.org/2017/1091.

[Cim15] Catalin Cimpanu. Epic fail: Power worm ransomware accidentally destroys victim's data during encryption. Available at http://news.softpedia.com/news/epic-fail-power-worm-ransomware-accidentally-destroys-victim-s-data-during-encryption-495833.shtml, 2015.

[DMMQN11] N. Döttling, T Mie, J. Müller-Quade, and T. Nilges. Basing obfuscation on simple tamper-proof hardware assumptions. In *TCC '11*. Springer, 2011.

[duk18] Duktape.org. Available at http://duktape.org, 2018.

[eth] The Ethereum Project. https://www.ethereum.org/.

[Eth17] Ethereum White Paper. Ethereum white paper. Available at https://github.com/ethereum/wiki/wiki/White-Paper, 2017.

[Fou] Apache Foundation. Hadoop Key Management Server (KMS) - Documentation Sets. Available at https://hadoop.apache.org/docs/stable/hadoop-kms/index.html.

[GG17a] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. Cryptology ePrint Archive, Report 2017/935, 2017. https://eprint.iacr.org/2017/935.

[GG17b] Rishab Goyal and Vipul Goyal. Overcoming cryptographic impossibility results using blockchains. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, pages 529–561, Cham, 2017. Springer International Publishing.

[GGH$^+$13] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. Cryptology ePrint Archive, Report 2013/451, 2013. `http://eprint.iacr.org/2013/451`.

[Gil15] Brett Giller. Implementing Practical Electrical Glitching Attacks. In *BlackHat '15*, 2015.

[GKW$^+$16] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 3–16, New York, NY, USA, 2016. ACM.

[Goo13] Dan Goodin. You're infected—if you want to see your data again, pay us $300 in bitcoins. `https://arstechnica.com/security/2013/10/youre-infected-if-you-want-to-see-your-data-again-pay-us-300-in-bitcoins/`, 2013.

[Hyp17] Hyperledger. Hyperledger Architecture, Volume 1. Available at `https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf`, 2017.

[Int] Intel Corporation. Hyperledger Sawtooth. Available at `http://hyperledger.org/projects/sawtooth`, 2018.

[Int16] Intel Corporation. Product Licensing FAQ. Available at `https://software.intel.com/en-us/sgx/product-license-faq`, 2016.

[Jag15] Tibor Jager. How to build time-lock encryption. Cryptology ePrint Archive, Report 2015/478, 2015. `http://eprint.iacr.org/2015/478`.

[JKS16] Ari Juels, Ahmed Kosba, and Elaine Shi. The ring of gyges: Investigating the future of criminal smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 283–295, New York, NY, USA, 2016. ACM.

[JSR$^+$16] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel Software Guard Extensions: EPID Provisioning and Attestation Services, 2016.

[Kan17] Michael Kan. Paying the WannaCry ransom will probably get you nothing. Here's why. *PCWorld*, 2017.

[Kau07] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *Usenix '07*, Berkeley, CA, USA, 2007. USENIX Association.

[KMG17] Gabriel Kaptchuk, Ian Miers, and Matthew Green. Managing secrets with consensus networks: Fairness, ransomware and access control. Cryptology ePrint Archive, Report 2017/201 (Revision 20170228:194725), 2017. `https://eprint.iacr.org/2017/201`.

[KMS$^+$16] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, May 2016.

[KRDO16] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. `https://eprint.iacr.org/2016/889`.

[Krs16] Ivan Krstić. Behind the Scenes with iOS Security. In BlackHat. Available at `https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf`, August 2016.

[Las17] LastPass. How is LastPass secure and how does it encrypt/decrypt my data safely? Available at `https://lastpass.com/support.php?cmd=showfaq&id=6926`, 2017.

[LMA+16] Kevin Lewi, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova. 5gen: A framework for prototyping applications using multilinear maps and matrix branching programs. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 981–992, New York, NY, USA, 2016. ACM.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[MAK+17] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. Cryptology ePrint Archive, Report 2017/048, 2017. `http://eprint.iacr.org/2017/048`.

[Nak08] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. 2008.

[NFR+17] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal. HOP: hardware makes obfuscation practical. In *NDSS '17*, 2017.

[opr18] Bitcoin wiki: Script. Available at `https://en.bitcoin.it/wiki/Script`, 2018.

[Pal15] Damian Paletta. FBI Chief Punches Back on Encryption. *Wall Street Journal*, July 2015.

[Pou01] Kevin Poulsen. DirecTV attacks hacked smart cards. *The Register*, 2001.

[Pro17] Android Project. Full-Disk Encryption. Available at `https://source.android.com/security/encryption/full-disk.html`, 2017.

[Rao16] Dinesh Rao. Intel SGX Product Licensing. Available at `https://software.intel.com/en-us/articles/intel-sgx-product-licensing`, 2016.

[Rog02] Phillip Rogaway. Authenticated encryption with associated data. In *CCS '02*. ACM Press, 2002.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *VIETCRYPT 2006*, pages 211–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[SA03] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *CHES '02*, pages 2–12, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[sgx] Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[Sin16] Denis Sinegubko. Website Ransomware - CBT-Locker Goes Blockchain. Sucuri Blog. Available at `https://blog.sucuri.net/2016/04/website-ransomware-ctb-locker-goes-blockchain.html`, April 2016.

[Sko16] Sergei Skorobogatov. The bumpy road towards iPhone 5c NAND mirroring. *CoRR*, abs/1609.04327, 2016.

[Tec16] Technology.org. Ransomware authors arrest cases. Available at `http://www.technology.org/2016/11/21/ransomware-authors-arrest-cases/`, November 2016.

[TLK+17] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. Obscuro: A bitcoin mixer using trusted execution environments. Cryptology ePrint Archive, Report 2017/974, 2017. `http://eprint.iacr.org/2017/974`.

[tpm] TPM Reset Attack. Available at `http://www.cs.dartmouth.edu/~pkilab/sparks/`.

[Tre16a] TrendMicro. Kansas hospital hit by ransomware, extorted twice. Available at http://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/kansas-hospital-hit-by-ransomware-extorted-twice, 2016.

[Tre16b] TrendMicro. Ransomware Update: UltraCrypter Not Giving Decrypt Keys After Payment, Jigsaw Changes UI Again. Available at http://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/ultracrypter-ransomware-no-decrypt-keys-jigsaw-changes-ui, 2016.

[USB+15] Blase Ur, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek, William Melicher, and Richard Shay. Measuring real-world accuracies and biases in modeling password guessability. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 463–481, Washington, D.C., 2015. USENIX Association.

[Wea15] Nicholas Weaver. iPhones, the FBI, and Going Dark. *Lawfare Blog*, August 2015.

[YY96] Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 129–140. IEEE, 1996.

[ZCC+16] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 270–282, New York, NY, USA, 2016. ACM.

[Zet16] Kim Zetter. Why hospitals are the perfect targets for ransomware. Available at https://www.wired.com/2016/03/ransomware-why-hospitals-are-the-perfect-targets/, 2016.

A Third-Party Privacy

We now provide a formal definition of *third-party privacy.* This definition is based on the following two experiments. In each experiment, we assume an honest enclave and an honest host, both configured via an execution of Setup (to obtain pp and K).

The third-party Privacy **Real** *experiment* ($\mathbf{Real}_{\mathsf{TPP}}$). This experiment is parameterized by a p.p.t. adversary $\mathcal{L}$ and an integer q. For $i = 1$ to q, the adversary $\mathcal{L}$ queries an oracle on a chosen program and user input (P_i, I_i). The oracle implements an honest host application combined with an honest enclave (both correctly configured with Setup), and for each distinct program P_i runs one step of the ExecuteApplication algorithm on the given input. The oracle provides $\mathcal{L}$ with the resulting data sent to the ledger. At the conclusion of the execution, the adversary $\mathcal{L}$ produces an output. This is the output of the experiment.

The third-party Privacy **Ideal** *experiment* ($\mathbf{Ideal}_{\mathsf{TPP}}$). This experiment is parameterized by a p.p.t. adversary $\mathcal{L}$ and an integer q, as well as a p.p.t. simulator algorithm $\mathcal{S}$. The experiment proceeds as follows. For $i = 1$ to q, the adversary $\mathcal{L}$ queries an oracle on chosen inputs (P_i, I_i). As in the **Real** experiment, the oracle implements an honest host application combined with an honest enclave and executes both functions on the given inputs. However, following the execution of the ExecuteApplication algorithm, the oracle runs the simulator $\mathcal{S}$ on input the public values $(\mathsf{pp}, \mathsf{Pub}_i)$. The output of $\mathcal{S}$ is provided to $\mathcal{L}$. At the conclusion of the experiment, $\mathcal{L}$ produces an output. This is the output of the experiment.

Definition A.1 (Third party privacy) *We say that a Enclave-Ledger scheme Σ_{eli} is* third party private *if $\forall$ p.p.t. real-world adversaries $\mathcal{L}$ and $\forall$ non-negative integers q, there exists a p.p.t. simulator $\mathcal{S}$ such that:*

$$\mathbf{Real}_{\mathsf{TPP}}(\Sigma_{\mathsf{eli}}, \mathcal{L}, q) \overset{c}{\approx} \mathbf{Ideal}_{\mathsf{TPP}}(\Sigma_{\mathsf{eli}}, \mathcal{L}, \mathcal{S}, q)$$

Remark. Note that this experiment implicitly provides privacy for user inputs, for the following reason. In the ideal-world experiment, the simulator $\mathcal{S}$ does not receive the user's input, output or the program P, and only takes the public output of the execution as input. Thus, if the two distributions are indistinguishable, the real protocol does not reveal more information than would naturally be revealed to the simulator by the public output.

Discussion. Although we do not present a complete security proof for the third party privacy of our main construction, here we present a brief intuition. Intuitively, note that each post made by the host application to the ledger consists of a commitment C embedding a user input. Assuming that the commitment scheme is computationally hiding, then no adversary will learn the contents of these commitments except with negligible advantage. This protects the confidentiality of user inputs against third parties.

B Proof of Theorem 3.1

Let $\mathcal{H}$ be an adversarial host that plays the **Real** experiment. We now construct an ideal-world adversarial host $\hat{\mathcal{H}}$ such that $\mathbf{Real}(\mathcal{H}, \lambda, q_{\mathsf{forge}}) \overset{c}{\approx} \mathbf{Ideal}(\hat{\mathcal{H}}, \lambda, q_{\mathsf{forge}})$ for sufficiently large λ and all non-negative q_{forge}.

Proof. To prove this statement, we first describe the operation of $\hat{\mathcal{H}}$. We then proceed to demonstrate that if there exists a p.p.t. distinguisher algorithm that distinguishes the output of the two experiments with non-negligible advantage, then one or more of the following are true: (1) there exists an adversary that breaks the $\mathsf{SUF\text{-}AUTH}$ security of the ledger authentication tags, (2) there is an attack on the binding property of the commitment scheme, (3) there exists a distinguisher for the pseudorandom function PRF, (4) there is an adversary that succeeds against the authenticated encryption scheme, or (5) there exists an attacker that breaks the collision-resistance of a hash function.

The operation of $\hat{\mathcal{H}}$. The ideal-world adversary $\hat{\mathcal{H}}$ runs the real adversary $\mathcal{H}$ internally, and simulates for it the oracles of the **Real** experiment. We will assume that $\hat{\mathcal{H}}$ also has access to an oracle that simulates the real ledger and produces authentication tags, in addition to a *ledger forgery* oracle that allows for the production of "forged" ledger signatures (see §2.2 for a discussion of this model.)

$\hat{\mathcal{H}}$ runs the **Ideal** experiment. Internally, it maintains the tables T_{enclave}, and T_{forge} as well as a view of the ledger L. The table T_{enclave} contains entries of the following form:

$$(P, i, \mathsf{I}_i, r_i, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathsf{post}, \mathsf{O}, \mathsf{Pub}, \mathsf{CID})$$

At the start of the experiment, our simulation generates $\mathsf{pp} \leftarrow \mathsf{CSetup}(1^{\lambda})$. Each query $\mathcal{H}$ makes to the **Real** oracles is handled as follows:

The Ledger Oracle. When $\mathcal{H}$ queries the ledger oracle on $(\mathsf{Data}, \mathsf{CID})$, $\hat{\mathcal{H}}$ executes $\mathsf{Ledger.Post}$ algorithm (see §2.2). It stores $(\mathsf{Data}, \mathsf{CID}, \sigma)$ in a table, and returns (post, σ) to $\mathcal{H}$. If two distinct outputs share the same value $\mathsf{post.Hash}$, the simulation aborts with an error.

The Forgery Oracle. This oracle responds to at most q_{forge} queries made by $\mathcal{H}$. When $\mathcal{H}$ queries on an arbitrary string S, $\hat{\mathcal{H}}$ queries the authentication oracle for a tag σ and records the pair (S, σ) in T_{forge}.

The Enclave Oracle. When $\mathcal{H}$ queries the enclave oracle on input $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)$, $\hat{\mathcal{H}}$ performs the following steps:

1. It performs all of the publicly-verifiable checks in the $\mathsf{ExecuteEnclave}$ algorithm (*i.e.,* all checks that do not rely on the secret K.) It will abort and output a defined error message if $\mathcal{H}$ outputs an authenticator/message pair that was not produced by the **Ledger** or **Forgery** oracles, a hash collision in H or $\mathsf{H_L}$, or a collision in the commitment scheme. $\mathsf{Event}_{\mathsf{hashcoll}}$.
2. It verifies that (post, σ) has either been posted to the ledger L, or exists in T_{forge}.
3. If $\mathcal{S}_i = \epsilon$ it looks up CID in the table T_{enclave} and aborts if any matching entry is found.

4. It searches T_{enclave} for an entry where $\mathcal{S}^*_{out} = \mathcal{S}_i$ and aborts if no result is found. Otherwise it obtains the tuple $(P^*, i^*, \mathsf{I}^*_i, r_i{}^*, \mathcal{S}^*_{in}, \mathcal{S}^*_{out}, \mathsf{post}^*, \mathsf{O}^*, \mathsf{Pub}^*, \mathsf{CID}^*)$ in T_{enclave} where $\mathcal{S}^*_{out} = \mathcal{S}_i$.
5. It checks that $i = i^* + 1$.
6. It checks that $\mathsf{CID} = \mathsf{post.CID} = \mathsf{post}^*.\mathsf{CID}$.
7. It checks that $\mathsf{post.PrevHash} = \mathsf{post}^*.\mathsf{Hash}$.
8. It checks that $\mathsf{post.Pub} = \mathsf{Pub}^*$.
9. It checks that $P = P^*$.

   If any of the above checks fail, $\hat{\mathcal{H}}$ aborts and outputs $\perp$ to $\mathcal{H}$.

10. $\hat{\mathcal{H}}$ first samples $k_{i+1} \leftarrow \{0,1\}^\ell$ uniformly at random, and computes a "dummy" ciphertext $\mathcal{S}'_{out} \leftarrow \mathsf{Encrypt}(k_{i+1}, (1^{\mathsf{Max}(P)}, 1^\ell))$.

    $\hat{\mathcal{H}}$ now selects one of the following three cases. The first case handles *repeated* inputs that have already been previously seen. The second case handles inputs that contain valid authenticators from the **Ledger** oracle. The final case handles inputs that have been authenticated by the **Forge** oracle.

11. **Process repeated program inputs.** Whenever $\hat{\mathcal{H}}$ receives an input, it searches its table for any entry that matches $(P, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$. (Note that we explicitly exclude σ_i from this check.[27]) If a matching entry exists, it obtains $(\mathsf{O}, \mathcal{S}_{\mathsf{out}}, \mathsf{Pub}_{\mathsf{out}})$ from the same entry in the table and outputs the tuple $(\mathcal{S}_{out}, \mathsf{O}, \mathsf{Pub}_{\mathsf{out}})$ to $\mathcal{H}$ and halts.
12. **Process new, and unforged inputs.** Otherwise, if $(\mathsf{post}_i, \sigma_i)$ is contained within the table maintained by the **Ledger** oracle, then $\hat{\mathcal{H}}$ looks up $\mathsf{post}_i.\mathsf{Hash}$ in the **Ledger**'s table to obtain CID. If this produces multiple possible matches, it aborts and outputs $\mathsf{Event}_{\mathsf{ledgerrepeat}}$.
    Next, $\hat{\mathcal{H}}$ calls the **Compute** oracle on $(P, \mathsf{I}_i, \mathsf{CID})$ to obtain $(\mathsf{O}', \mathsf{Pub}')$. $\hat{\mathcal{H}}$ now stores $(P, i, \mathsf{I}_i, r_i, \mathcal{S}_i, \mathcal{S}'_{out}, \mathsf{post}, \mathsf{O}', \mathsf{Pub}')$ in T_{enclave}. It returns $(\mathcal{S}'_{out}, \mathsf{O}', \mathsf{Pub}')$ to $\mathcal{H}$.
13. **Process new, forged inputs.** Otherwise, if (post, σ) is contained within T_{forge}, then $\hat{\mathcal{H}}$ identifies the call j at which this value was added to the table, and calls the **Fork** oracle on input $(P, \mathsf{I}_, \mathsf{CID}, j)$ to obtain $(\mathsf{O}', \mathsf{Pub}')$. As in the previous step, it stores $(P, i||j, \mathsf{I}_i, r_i, \mathcal{S}_i, \mathcal{S}'_{out}, \mathsf{post}, \mathsf{O}', \mathsf{Pub}', \mathsf{CID})$ in T_{enclave}. It returns $(\mathcal{S}'_{out}, \mathsf{O}', \mathsf{Pub}')$ to $\mathcal{H}$.

Discussion. Note that there are three main cases in the simulation above. Whenever the adversary queries the enclave on an state/counter that has previously been queried, $\hat{\mathcal{H}}$ simply returns the same output as it did during the previous query (this occurs at step 11). This is possible because the output of the enclave in the protocol is determined solely by the given inputs (if all public checks pass), and thus repeated inputs cause the enclave to produce the same behavior.

When the adversary queries on a new input that has been posted to the ledger, $\hat{\mathcal{H}}$ queries the ideal **Compute** oracle to obtain the appropriate output, and generates a simulated ("dummy ciphertext") encrypted state to return to $\mathcal{H}$. Similarly, when $\mathcal{H}$ queries the enclave on a forged (but otherwise correct) ledger output, $\hat{\mathcal{H}}$ queries the **Fork** oracle to obtain the correct output and also returns a dummy ciphertext to $\mathcal{H}$.

Indistinguishability of $\hat{\mathcal{H}}$'s simulation. Let D be a p.p.t. distinguisher that succeeds in distinguishing $\hat{\mathcal{H}}$'s output in the **Ideal** experiment from $\mathcal{H}$'s output in the **Real** experiment with non-negligible advantage. We now show that such an adversary violates one of our assumptions above. The proof proceeds via a series of hybrids, where in each hybrid $\mathcal{H}$ interacts as in the **Real** experiment. The first hybrid (**Game 0**) is identically distributed to the **Real** experiment, and the final hybrid represents $\hat{\mathcal{H}}$'s simulation above. For notational convenience, let **Adv** [**Game i**] be D's absolute advantage in distinguishing the output of **Game i** from **Game 0**, *i.e.,* the **Real** distribution.

Game 0. In this hybrid, $\mathcal{H}$ interacts with the **Real** experiment.

[27]Recall that our simulation has already verified that σ_i passes the value passes the check $\mathsf{Assert}(\mathsf{Ledger.Verify}(\mathsf{post}_i, \sigma_i))$. Provided that this check succeeds, it is easy to see that the value σ_i has no further influence on the output of $\mathsf{ExecuteEnclave}$.

Game 1 (**Abort on [adversary-]forged authenticators.**) This hybrid modifies the previous as follows: in the event that $\mathcal{H}$ queries the **Enclave** oracle on any pair (post, σ) such that (1) $\mathsf{Ledger.Verify}(\mathsf{post}, \sigma) = 1$, and yet (2) the pair was not the input (resp. output) of a previous call to either the **Ledger** or **Forgery** oracles, then abort and output $\mathsf{Event}_{\mathsf{forge}}$. We note that if $\mathcal{H}$ causes this event to occur with non-negligible probability, then we can trivially use $\mathcal{H}$ to construct an attack on the $\mathsf{SUF\text{-}AUTH}$ security of the contract authenticator with related probability. Since by assumption the probability of such an event is negligible, we bound $\textbf{Adv}\,[\,\textbf{Game 1}\,] \leq \nu_1(\lambda)$.

Game 2 (**Abort on hash collisions.**) This hybrid modifies the previous as follows: if at any point during the experiment $\mathcal{H}$ causes the functions $\mathsf{H}, \mathsf{H_L}$ to be evaluated on inputs $s_1 \neq s_2$ such that $\mathsf{H}(s_1) = \mathsf{H}(s_2)$ or $\mathsf{H_L}(s_1) = \mathsf{H_L}(s_2)$, then abort and output $\mathsf{Event}_{\mathsf{hashcoll}}$. Under the assumption that the hash functions $\mathsf{H}, \mathsf{H_L}$ are collision-resistant, we have that $|\textbf{Adv}\,[\,\textbf{Game 2}\,] - \textbf{Adv}\,[\,\textbf{Game 1}\,]| \leq \nu_2(\lambda)$.

Game 3 (**Abort on commitment collisions.**) This hybrid modifies the previous as follows: in the event that $\mathcal{H}$ queries the **Enclave** oracle at steps i, j where $C_i = C_j = \mathsf{Commit}(\mathsf{pp}, (i, \mathsf{I}_i, \mathcal{S}_i, P, \mathsf{post}_i.\mathsf{CID}); r_i) = \mathsf{Commit}(\mathsf{pp}, j \| \mathsf{I}_j \| \mathcal{S}_j \| \mathsf{post}_i.\mathsf{CID}; r_j)$ and yet $(i, \mathsf{I}_i, \mathcal{S}_i, P_i, \mathsf{post}_i.\mathsf{CID}) \neq (j, \mathsf{I}_j, \mathcal{S}_j, P_j, \mathsf{post}_j.\mathsf{CID})$, then abort and output $\mathsf{Event}_{\mathsf{binding}}$. We note that if the commitment scheme is binding, this event will occur with at most negligible probability, hence $|\textbf{Adv}\,[\,\textbf{Game 3}\,] - \textbf{Adv}\,[\,\textbf{Game 2}\,]| \leq \nu_3(\lambda)$.

Game 4 (**Duplicate Enclave calls give identical outputs.**) This hybrid modifies the previous as follows: when $\mathcal{H}$ queries the **Enclave** oracle repeatedly on the same values $(P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$ (here we exclude σ_i) and the oracle (as implemented in the previous hybrid) does not output $\perp$, replace the response to all repeated queries subsequent to the first query with the same result as the first query. Recall that by definition the **Enclave** oracle's $\mathsf{ExecuteEnclave}$ calculation is deterministic and solely based on the inputs provided above. Thus, repeated queries on the same input will always produce the same output. Hence by definition $|\textbf{Adv}\,[\,\textbf{Game 4}\,] - \textbf{Adv}\,[\,\textbf{Game 3}\,]| = 0$.

Game 5 (**Abort on colliding ledger hashes.**) This hybrid modifies the previous as follows: when $\mathcal{H}$ calls the **Enclave** oracle on two distinct inputs $(P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i, \sigma_i)$ and $(P_j, j, \mathsf{I}_j, \mathcal{S}_j, \mathsf{post}_j, \sigma_j)$, and if the two inputs do *not* represent repeated inputs (according to **Game 4**), then: if both $(\mathsf{post}_i, \sigma_i)$ and $(\mathsf{post}_j, \sigma_j)$ are valid outputs of the **Ledger** oracle and yet $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ then abort and output $\mathsf{Event}_{\mathsf{ledgercoll}}$. By Lemma B.1 this event will occur with at most negligible probability if $\mathsf{H_L}$ is collision-resistant and the ledger is implemented correctly. This bounds $|\textbf{Adv}\,[\,\textbf{Game 5}\,] - \textbf{Adv}\,[\,\textbf{Game 4}\,]| \leq \nu_4(\lambda)$.

Game 6 (**$\hat{\mathcal{H}}$ can always uniquely identify CID.**) This hybrid modifies the previous as follows: if at step i the adversary $\mathcal{H}$ calls the **Enclave** the oracle (as implemented in the previous hybrid) and (1) the oracle does not return $\perp$, (2) the inputs to the two calls are not identical (this would be excluded by the earlier hybrids), and (3) the pair $(\mathsf{post}_i, \sigma_i)$ are in the **Ledger** table, and (4) $\mathsf{post}_i.\mathsf{Hash}$ matches two distinct entries in the **Ledger** table, then abort and output $\mathsf{Event}_{\mathsf{ledgerrepeat}}$. By Lemma B.2 this event will occur with at most negligible probability. This bounds $|\textbf{Adv}\,[\,\textbf{Game 6}\,] - \textbf{Adv}\,[\,\textbf{Game 5}\,]| \leq \nu_5(\lambda)$.

Game 7 (**Replace the session keys and pseudorandom coins with random strings.**) This hybrid modifies the previous as follows: if **Enclave** does not abort or is not called on repeated inputs, then the pair $(k_{i+1}, \bar{r}_i)$ is sampled uniformly at random and recorded in a table for later use. Recall that in the preceding hybrid, this pair is generated as $(k_{i+1}, \bar{r}_i) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$ where (by the previous hybrids) $\mathsf{post}_i.\mathsf{Hash}$ is guaranteed not to repeat. Similarly, the output of each call $(k_i, \cdot) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{PrevHash})$ is also replaced with a random value when the input $\mathsf{post}_i.\mathsf{PrevHash}$ has not been queried to PRF_K previously, or the appropriate value (drawn from the table) when this value has been previously queried. If PRF is pseudorandom then if $|\textbf{Adv}\,[\,\textbf{Game 5}\,] - \textbf{Adv}\,[\,\textbf{Game 4}\,]|$ is non-negligible, then we can construct an algorithm that succeeds in distinguishing the PRF from a random function. This bounds $|\textbf{Adv}\,[\,\textbf{Game 7}\,] - \textbf{Adv}\,[\,\textbf{Game 6}\,]| \leq \nu_6(\lambda)$.

Game 8 (**Reject inauthentic ciphertexts.**) This hybrid modifies the previous as follows: if $\mathcal{H}$ queries the **Enclave** oracle on an input $\mathcal{S}_i \neq \varepsilon$ such that (1) the oracle does not reject the input, (2) $\mathsf{Decrypt}(k_i, \mathcal{S}_i)$ does not output $\perp$, and yet (3) the pair $(\mathcal{S}_i, k_i)$ was not generated during a previous query to **Enclave**, then abort and output $\mathsf{Event}_{\mathsf{auth}}$. We note that that in the previous hybrid each key k_i is generated at random, and only used to encrypt one ciphertext. Thus if $\mathcal{H}$ is able to produce a second ciphertext that satisfies decryption under this key, then we can construct an attacker that forges ciphertexts in the authenticated encryption scheme with non-negligible advantage. Because we assumed the AE scheme was secure, we have that $|\mathbf{Adv}[\,\mathbf{Game\ 8}\,] - \mathbf{Adv}[\,\mathbf{Game\ 7}\,]| \le \nu_7(\lambda)$.

Game 9 (**Abort if inputs are inconsistent.**) This hybrid modifies the previous as follows: on $\mathcal{H}$'s the i^{th} query to the **Enclave** oracle, when the input $\mathcal{S}_i \neq \varepsilon$, let $(\mathsf{post.CID}', P', i', \mathsf{Pub}_{i-1}{}')$ be the inputs/outputs associated with the previous **Enclave** call that produced $\mathcal{S}_i$. If (1) the experiment has not already aborted due to a condition described in previous hybrids and (2) if the **Enclave** oracle as implemented in the previous hybrid does not reject the input, and (3) any of the provided inputs $(\mathsf{post.CID}, P, i-1, \mathsf{Pub}_{i-1}) \neq (\mathsf{post.CID}', P', i', \mathsf{Pub}_{i-1}{}')$ differ from those associated with the previous call to the **Enclave**, abort and output $\mathsf{Event}_{\mathsf{mismatch}}$. By Lemma B.3 we have that $|\mathbf{Adv}[\,\mathbf{Game\ 9}\,] - \mathbf{Adv}[\,\mathbf{Game\ 8}\,]| = 0$.

Game 10 (**Replace ciphertexts with dummy ciphertexts.**) This hybrid modifies the previous as follows: we modify the generation of each ciphertext $\mathcal{S}'_{out}$ to encrypt the unary string $(1^{\mathsf{Max}(P)}, 1^{\ell})$. We first note that the length of the (padded) plaintext is identical between this and the previous hybrid, by the definition of the padding function. Thus if $\mathcal{H}$'s output is significantly different between this and the previous hybrid, we can construct an attacker that succeeds against the confidentiality of the authenticated encryption scheme. By the assumption that the AE scheme is secure, we have that $|\mathbf{Adv}[\,\mathbf{Game\ 10}\,] - \mathbf{Adv}[\,\mathbf{Game\ 9}\,]| \le \nu_8(\lambda)$.

We note that **Game 10** is distributed identically to the simulation provided to $\mathcal{H}$ by $\hat{\mathcal{H}}$. By summation over the hybrids we have that $\mathbf{Adv}[\,\mathbf{Game\ 9}\,] \le \nu_1(\lambda) + \cdots + \nu_8(\lambda)$ and thus is also negligible in λ. This implies that no p.p.t. distinguisher can distinguish the output of **Real** and **Ideal** experiments with non-negligible advantage.

□

Lemma B.1 (Uniqueness of Ledger Identifiers) If $\mathsf{H_L}$ is collision resistant and the ledger operates as described in §2.2, then for *non-repeated* inputs (those not caught in **Game 4**) with all but negligible probability no two calls $i \neq j$ to the **Enclave** oracle in **Game 5** will have $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$.

Proof. Let us imagine that with some non-negligible probability ϵ, $\mathcal{H}$ is able to play **Game 5** such that two distinct calls $i \neq j$ to the **Enclave** oracle have $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ and yet the two calls were not identified as repeated inputs (according to the conditions of **Game 4**). We show that this implies a collision in the hash function $\mathsf{H_L}$.

Let $(P_j, j, \mathsf{I}_j, \mathcal{S}_j, \mathsf{post}_j), (P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$ be the inputs to the **Enclave** oracle (we exclude σ). Let $(\mathsf{Data}_i, \mathsf{CID}_i, \mathsf{post}_i.\mathsf{Hash}), (\mathsf{Data}_j, \mathsf{CID}_j, \mathsf{post}_j.\mathsf{Hash})$ represent $\mathcal{H}$'s input (resp. output) from the distinct calls to the **Ledger** oracle. We now show that if two such calls produce identical $\mathsf{post}_i.\mathsf{Hash}$, then the inputs to $\mathsf{H_L}$ are not equal with probability related to ϵ.

If the two calls to **Enclave** are *not* repeated inputs, then for $i \neq j$ it holds that:

$$(P_j, j, \mathsf{I}_j, \mathcal{S}_j, \mathsf{post}_j) \neq (P_j, j, \mathsf{I}_j, \mathcal{S}_j, \mathsf{post}_j)$$

Recall as well that:

$$\mathsf{post}_i.\mathsf{Hash} = \mathsf{H_L}(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash}) \text{ and } \mathsf{post}_j.\mathsf{Hash} = \mathsf{H_L}(\mathsf{post}_j.\mathsf{Data} \| \mathsf{post}_j.\mathsf{PrevHash})$$

And:

$$\mathsf{post}_i.\mathsf{Data} = \mathsf{Commit}(\mathsf{pp}, (i, \mathsf{I}_i, \mathcal{S}_i, P_i); r_i) \text{ and } \mathsf{post}_j.\mathsf{Data} = \mathsf{Commit}(\mathsf{pp}, (j, \mathsf{I}_j, \mathcal{S}_j, P_j); r_j)$$

Thus note that if $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ then this would imply that the simulation would abort with either $\mathsf{Event}_{\mathsf{hashcoll}}$ or $\mathsf{Event}_{\mathsf{binding}}$. Since this has not occurred, then the probability of such a collision is 0. □

Lemma B.2 (No duplicate Ledger identifiers) If H_L is collision-resistant, then for all $\mathcal{H}$, $\mathbf{Pr}\left[\,\mathsf{Event}_{\mathsf{ledgerrepeat}}\,\right] \leq \nu_6(\lambda)$.

Proof. Let us assume by contradiction that $\mathcal{H}$ is able to query the **Ledger** oracle on two distinct inputs such that the oracle returns (and records in its table) two distinct transactions post_i and post_j such that $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$. Then we construct a second adversary $\mathcal{A}$ that outputs a collision in the hash function H_L. $\mathcal{A}$ conducts the experiment of **Game 6** with $\mathcal{H}$.

Let i, j be any two integers. If at any point $\mathcal{H}$, at the j^{th} call to the **Ledger** oracle, submits a pair $(\mathsf{Data}, \mathsf{CID})$ such that $\mathsf{post}_j.\mathsf{Hash} = \mathsf{post}_i.\mathsf{Hash}$, then $\mathcal{A}$ terminates and outputs the collision pair $(\mathsf{post}_i.\mathsf{Data}\|\mathsf{post}_i.\mathsf{PrevHash}), (\mathsf{post}_j.\mathsf{Data}\|\mathsf{post}_j.\mathsf{PrevHash})$.

Clearly if $\mathsf{post}_j.\mathsf{Hash} = \mathsf{post}_i.\mathsf{Hash}$ and yet $(\mathsf{post}_i.\mathsf{Data}\|\mathsf{post}_i.\mathsf{PrevHash}) \neq (\mathsf{post}_j.\mathsf{Data}\|\mathsf{post}_j.\mathsf{PrevHash})$ then $\mathcal{A}$ has identified a collision in H_L. It remains only to show that when $\mathcal{H}$ succeeds in triggering this event, the latter inequality must hold. Our proof proceeds inductively.

1. If $j = 0$ then this condition cannot occur, as there is no previous entry in the table.
2. If $j > 0$ then there are two subconditions:
   (a) If $\mathcal{H}$ has *not* previously called the **Ledger** oracle on CID then $\mathsf{post}_j.\mathsf{PrevHash}$ is set to a unique identifier based on CID. Because there has been no previous call on input CID, there cannot exist a second value $\mathsf{post}_i.\mathsf{PrevHash}$ in the table that shares the same value. Thus $\forall i$ it holds that $\mathsf{post}_j.\mathsf{PrevHash} \neq \mathsf{post}_i.\mathsf{PrevHash}$ and thus the main inequality holds.
   (b) If $\mathcal{H}$ has previously called the **Ledger** oracle on input CID, then (by the definition of the Ledger interface) there must exist some i such that $\mathsf{post}_j.\mathsf{PrevHash} = \mathsf{post}_i.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_i.\mathsf{Data}\|\mathsf{post}_i.\mathsf{PrevHash})$. We now consider two subcases:
      i. If the i^{th} call to the **Ledger** oracle was the first call made on input CID, then by definition $\mathsf{post}_i.\mathsf{PrevHash} \neq \mathsf{post}_j.\mathsf{PrevHash}$ because as the root of a new chain, $\mathsf{post}_i.\mathsf{PrevHash}$ has a special structure and cannot be equal to the output of H_L.
      ii. If the i^{th} call to **Ledger** was *not* the first call on input CID, and if $\mathsf{post}_j.\mathsf{PrevHash} = \mathsf{post}_i.\mathsf{PrevHash}$, then by definition there must exist a third integer $k < j$ such that the k^{th} call to the **Ledger** oracle was also on input CID and $\mathsf{post}_i.\mathsf{PrevHash} = \mathsf{post}_k.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_k.\mathsf{Data}\|\mathsf{post}_k.\mathsf{PrevHash})$. However, this event cannot occur, as this would imply that $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_k.\mathsf{Hash}$ and thus $\mathcal{A}$ would have already terminated and output a collision prior to reaching this point.

In all of the above cases it holds that $(\mathsf{post}_i.\mathsf{Data}\|\mathsf{post}_i.\mathsf{PrevHash}) \neq (\mathsf{post}_j.\mathsf{Data}\|\mathsf{post}_j.\mathsf{PrevHash})$ and so if $\mathcal{H}$ triggers this condition, then $\mathcal{A}$ finds a collision in H_L. Because we assume that H_L is collision-resistant, we can bound the probability of $\mathsf{Event}_{\mathsf{ledgerrepeat}}$ to be negligible in λ.

□

Lemma B.3 (Consistency of encrypted state) It holds that for all p.p.t. $\mathcal{H}$, $\mathbf{Pr}\left[\,\mathsf{Event}_{\mathsf{mismatch}}\,\right] = 0$.

Proof. For $\mathsf{Event}_{\mathsf{mismatch}}$ to occur, one of the following conditions must occur. First, it must be the case that $\mathcal{H}$ has previously called **Enclave** on a set of values $(\mathsf{post}.\mathsf{CID}', P', i', \mathsf{Pub}_{i-1}{}')$, which produced a state ciphertext $\mathcal{S}$ that embeds $H_P = \mathsf{H}(\mathsf{post}.\mathsf{CID}'\|P'\|i'\|\mathsf{Pub}'_{i-1})$. And simultaneously it must be the case that, on input the state ciphertext $\mathcal{S}$, the assertion $\mathsf{Assert}(H_P = \mathsf{H}(\mathsf{CID}\|P\|i\|\mathsf{Pub}_{i-1}))$ would not fail and cause the oracle to return $\perp$.

However, for this event to occur, it would require one of the following to be true: either the state ciphertext would have to be inauthentic, resulting in a previous abort $\mathsf{Event}_{\mathsf{auth}}$ (or another abort). Or the hash function would have to include a collision, resulting in $\mathsf{Event}_{\mathsf{hashcoll}}$. Since we require that the simulation would abort on these other events before it aborts with $\mathsf{Event}_{\mathsf{mismatch}}$, this event can never occur. □

Lemma B.4 (Uniqueness of PRF inputs) No adversary playing **Game 5** will (with non-negligible probability) call the **Enclave** oracle at steps i, j that (1) the oracle does not return $\perp$ or the experiment aborts in one of the calls, (2) the input values $(P_1, i, \mathsf{I}_1, \mathcal{S}_1, \mathsf{CID}_1)$ and $(P_2, j, \mathsf{I}_2, \mathcal{S}_2, \mathsf{CID}_2)$ to the two oracle calls are distinct, and (3) during these invocations the evaluation $\mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$ and $\mathsf{PRF}_K(\mathsf{post}_j.\mathsf{Hash})$ uses $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$.

Proof. Let $\mathcal{H}'$ be an adversary that succeeds in triggering the above condition with non-negligible probability ϵ. We now show this violates one of the assumptions given above.

Let us define the two (partial) input tuples provided to the **Enclave** oracle as $\mathsf{Input}_1 = (P_1, i, \mathsf{I}_1, \mathcal{S}_1, \mathsf{CID}_1)$ and $\mathsf{Input}_2 = (P_2, j, \mathsf{I}_2, \mathcal{S}_2, \mathsf{CID}_2)$ where $\mathsf{Input}_1 \neq \mathsf{Input}_2$ (we also separately define $\mathsf{post}_1, \mathsf{post}_2$ as inputs to the oracle.) Recall that in the first instance, PRF is evaluated on $(K, \mathsf{post}_i.\mathsf{Hash})$ and in the second it is evaluated on $(K, \mathsf{post}_j.\mathsf{Hash})$. We now argue that if the oracle does not output $\perp$ (and the experiment itself has not aborted), and if $\mathsf{Input}_i \neq \mathsf{Input}_j$ then this must imply that $\mathsf{post}_i.\mathsf{Hash} \neq \mathsf{post}_j.\mathsf{Hash}$.

Note that each of the fields $\mathsf{post}_1.\mathsf{Data}, \mathsf{post}_2.\mathsf{Data}$ embeds a commitment computed on $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1)$, $(\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ respectively. If the **Enclave** oracle does not abort and the experiment does not abort, then if $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1) \neq (\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ then it must hold that (1) $\mathsf{post}_1.\mathsf{Data} \neq \mathsf{post}_2.\mathsf{Data}$ (if this were not the case then it would imply a violation of the binding property of the commitment scheme, and hence an abort with $\mathsf{Event}_{\mathsf{binding}}$), or (2) $\mathsf{post}_1.\mathsf{Hash} \neq \mathsf{post}_2.\mathsf{Hash}$ (because each Hash is computed by applying $\mathsf{H_L}$ to the Data field, and a collision in these values would imply an abort with $\mathsf{Event}_{\mathsf{hashcoll}}$).

Since we required that these aborts do *not* occur, it must hold that if $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1) \neq (\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ then $\mathsf{post}_i.\mathsf{Hash} \neq \mathsf{post}_j.\mathsf{Hash}$. This concludes the proof. $\square$