Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers

Gabriel Kaptchuk, Matthew Green
Johns Hopkins University
{gkaptchuk, mgreen}@cs.jhu.edu

Ian Miers
Cornell Tech
imiers@cs.cornell.edu

Abstract—In this work we investigate new computational properties that can be achieved 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.

I. INTRODUCTION

In recent years a new class of distributed system has evolved. Loosely categorized as decentralized ledgers, these distributed systems construct a virtual “bulletin board” to which nodes may publish data. Many protocols, including cryptocurrencies such as Bitcoin [47], construct such a ledger to record financial transactions. More recent systems target other specific applications, such as identity management [30], [43], the execution of general, user-defined programs, called “smart contracts” [6]. Some companies have deployed centralized public ledgers for specific applications; for example, Google’s Certificate Transparency [3] provides a highly-available centralized ledger for recording issued TLS certificates.

Regardless of the intended application, two facts seem clear: (1) centralized and decentralized ledger systems are already in widespread deployment, and this deployment is likely to continue. Moreover (2) the decentralized nature of these systems makes them potentially long-lived and resilient to certain classes of network-based attack. This provides a motivation to identify new ways that these technologies can be used to enhance the security of distributed systems.

In this work we focus on one such application: using ledgers to enhance the security of Trusted Execution Environments (TEE). In the context of this work, we use TEE to refer to any limited, secure computing environment that is dependent on a (possibly malicious) host computer for correct operation. Examples of such environments include Hardware Security Modules, smart cards [51], and the “secure element” co-processors present in many mobile devices [12], as well as virtualized TEE platforms such as Intel’s Software Guard Extensions (SGX), ARM TrustZone, and AMD SEV [14], [15]. While these examples rely on hardware, it is conceivable that future trusted environments may be implemented using pure software virtualization, general-purpose hardware obfuscation [26], [23], [48], or even cryptographic program obfuscation [43].

While TEEs have many applications in computing, they (and all secure co-processors) have fundamental limitations. A trusted environment operating perfectly depends on the host computer for essential functionality, creating an opportunity for a malicious host to manipulate the TEE and its view of the world. For example, an attacker may:

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

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

We stress that these attacks may have a catastrophic impact even if the TEE 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 [17], [7], [27]. State reset attacks are not merely a problem for cryptographic protocols; they are catastrophic for many typical applications such as limited-attempt password checking [57].

When implemented in hardware, TEE systems can mitigate reset attacks by deploying a limited amount of tamper-resistant non-volatile storage [50]. However, such countermeasures...
increase the cost of producing the hardware and are simply not possible in software-only environments. Moreover, these countermeasures are unavailable to environments where a single state transition machine is run in a distributed fashion, with the transition function executed across different machines. In these environments, which include private smart contract systems [31] and “serverless” cloud step-function environments [9], [28], state protections cannot be enforced locally. Similarly, no degree of hardware support can 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 [44]. These could keep state on behalf of the enclave and would act as conduit to the public network. However, this approach simply shifts the root of trust to a different physical location, failing to solve our problem because this new system is 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 TEEs 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 TEE to include secure internal non-volatile storage, nor does it require a protocol-aware external server to keep state. Instead, we propose a 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 received ledger data is authentic. The main security requirement we require from the ledger is that its contents cannot be modified or erased and proofs of publication cannot be (efficiently) forged. This model has been previously investigated independently in a more limited fashion by other works, notably in the context of fair multiparty computation [22], [29]. In this work we propose a broader paradigm for secure computation.

**Our contributions.** 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 TEE enclave (for the rest of this work, we use these two terms interchangeably) 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, SGX enclave, or a cryptographically obfuscated circuit [43]. Most notably, the enclave need not store persistent state or possess a secure random number 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 enclave-ledger interaction allows us to condition program execution on the publication of particular messages to the ledger, or the receipt of messages from third parties. For example, an application can require that developers be alerted on the ledger that user activity is anomalous, perhaps even dangerous, before it continues execution.

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 [57]), enforced file access logging, and new forms of encryption that ensure monetary payments 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.

**Previous and concurrent work.** A manuscript of this work was initially published in February 2017 [29]. Several previous and concurrent works have focused on similar goals, specifically preventing rollback attacks on trusted execution environments and private computation on the ledger.

In previous work, Memoir [50] leverages hashchains, NVRAM, and monotonic counters to efficiently prevent state rollback in the presence of a malicious host. While Memoir’s protocol has many similarities to our ELI protocols, the system design is quite different. In Memoir, each TEE device uses its internal NVRAM to checkpoint state, while our systems rely on a public ledger and communication through an untrustworthy host. A second proposal, ROTE [44], uses a consensus between a cluster of distributed enclaves to in order to address the rollback problem. Neither Memoir nor ROTE deals with the problem of conditioning execution on public data, which is a second contribution of our work. Several older works have focused on the problem of preventing reset attacks via de-randomization, i.e., by deriving (pseudo)random coins from the computation input [20]. Unfortunately this approach does not generalize to multi-step calculations where the adversary can adaptively select the input prior to each step.

Two concurrent research efforts have considered the use of ledgers to achieve secure computation. In late 2017, Goyal and Goyal proposed the use of blockchains for implementing one time programs [29] 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. Also in 2017, the authors of the present work, along with others, proposed to use ledgers to obtain fairness for MPC protocols, an application that is discussed in later sections of this work [22]. Finally, Bowman et al. of Intel Corporation [19] independently proposed “Private Data Objects” for smart contract systems that use ideas related to this work, and have begun to implement them in production smart contract systems that support private computation. 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 [40], [36], [65]. Finally, the Ekiden system [21], proposed in April 2018, builds on the ideas proposed in this work and [22] to achieve goals similar to those of Intel’s Private Data Objects.

A. Intuition

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

$$P(l_i, S_i; r_i) \rightarrow (O_i, S_{i+1})$$

At each step of the program execution, the program takes a user input $l_i$, an (optional) state $S_i$ from the previous execution step, along with some random coins $r_i$. 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 $\sigma$. We also require a stateless enclave with no native random number generator, that stores a single, hardcoded, secret key $K$.

Figure 1 illustrates the way the user, host, ledger and enclave can interact. We now discuss several candidate approaches, beginning with obviously insecure ideas, and building on them to describe 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.

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 future execution of $P$ on receiving valid ledger contents $L$, as well as a proof of publication $\sigma$, and extracting the encrypted state from $L$.

Unfortunately this does nothing to solve the problem of adversarial replays. Because the enclave has no trusted source of time and 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 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 $l_i$ to the ledger. This can be

---

2In later sections we will discuss improvements that make this Ledger response succinct.

3For 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.

---
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 $\sigma$. 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 $\epsilon$ if this is the first step of the program). The enclave decrypts the encrypted state internally to obtain the program state and counter $(S, i)$.

It verifies the following conditions:

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

If all conditions are met, the enclave can now execute the program on state and input $(S, 1)$. Following execution, it encrypts the new output state and updated counter $(S_{i+1}, i+1)$ 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, 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.\footnote{If no encrypted state is provided, then $i$ is implicitly set to 0 and $S = \epsilon$.}

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 from the inputs, using a pseudorandom function (similar to the classical approach of Canetti et al. [20]). This approach fixes the random coins used at each computation step and effectively binds them to the ledger and the host’s chosen input.

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. We discuss several extensions below.

Extension #1: Reducing ledger bandwidth. The pedagogical protocol above requires the host and enclave to parse the entire ledger $L$ on each execution step. This is quite impractical, especially for public ledgers that may contain millions of transactions. A key contribution of this work is to show that the enclave need not receive the entire ledger contents, provided that the ledger can be given only 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 [II-B] and [V-B] these capabilities are already present in many candidate ledger systems such as public (and private) blockchain networks.

Extension #2: Adding public input and output. A key goal of our protocol is to allow $P$ to condition its execution on inputs and outputs drawn from (resp. sent to) the ledger. This can be achieved because due the enclave receives an authenticated copy of $L$. Thus the enclave (and $P$) can be designed to condition its operation on e.g., messages or public payment data found on the ledger.

To enforce public output, we modify the interface of $P$ to produce a “public output string” as part of its output to the host, and we record this string with the program’s encrypted state. Due to the program’s encrypted state, the host 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 if the host does not comply with the protocol.

Extension #3: 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.

Modeling the ledger Several recent works have also used ledgers (or bulletin boards) to provide various security properties [29, 22]. In these works, the ledger is treated as possessing an unforgeable proof of publication. The protocols in this work can operate under this assumption, however our construction is also motivated by real-world decentralized ledgers, many of which do not have possess such a property. Instead, many “proof-of-work” blockchains provide a weaker security property, in that it is merely expensive to forge a proof that a message has been posted to the blockchain. This notion may provide sufficient security in many real-world applications, and we provide a detailed analysis of the costs in [V-B].
B. 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. Here we provide several example applications, and provide a more complete discussion in §V

Synchronizing private smart contracts and step functions. Smart contract systems and cloud “step functions” [24], [31] each employ a distributed network of compute nodes that perform a multi-step interactive computation. To enable private computation, some production smart contract systems [31] have recently proposed incorporating TEEs. Such distributed systems struggle to synchronize state as the computation progresses across nodes. Motivated by an independent effort of Bowman et al. [19] we show that our ELI paradigm achieves the necessary guarantees for security in this setting.

Mandatory logging for local file access. Corporate and enterprise settings often require users to log access to sensitive files, usually on 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.

Limiting password guessing. Cryptographic access control systems often employ passwords to control access to encrypted filesystems [12], [52] and cloud backup images (e.g., Apple’s iCloud Keychain [41]). 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 [13], [61]⁶. Attempts to address this with tamper-resistant hardware [12], [52], [14], [41] lead to expensive systems that provide no security against rewind attacks.⁷ We show that ELI can safely enforce passcode guessing limits using only inexpensive hardware without immutable state [57].

Autonomous ransomware. Modern ransomware, malware that encrypts a victim’s files, is tightly integrated with cryptocurrencies such as Bitcoin, which act as both the ransom currency and a communication channel to the attacker [56]. Affected users must transmit an encrypted key package along with a ransom payment to the attacker, who responds with the necessary decryption keys. The ELI paradigm could potentially enable 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.

II. DEFINITIONS

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

The ledger 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 σi over the entire ledger contents, or a portion of the ledger.

The enclave 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 H, it calculates and returns data to the host.

The host application 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.

A. 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 $P : \mathbb{Z} \times \mathbb{S} \times \mathcal{R} \rightarrow \mathbb{O} \times \mathbb{P} \times \mathbb{S}$ that has the following input/output interface:

$$P(l_i, S_i; \bar{r}_i) \rightarrow (O_i, Pub_i, S_{i+1}).$$

When a user inputs $l_i$, 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 $S_{i+1}$.

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 $\text{Max}()$ such that $\text{Max}(P)$ indicates the maximum length in bits of any output state $S_i$ produced by $P$, and that $\text{Max}(P)$ 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 interleaved. 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 identified by a session identifier, which we denote by $\text{CID}$. 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.

### B. 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.

The pedagogical examples examined thus far have been very bandwidth-intensive, potentially transmitting the entire contents of the ledger with each authentication tag. Such implementations would be impractical, especially 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 naive ledger that returns the full ledger contents. As such we are not truly adding new requirements.)

**The Ledger Interface.** Our ideal ledger incorporates an arbitrary string $S$ as part of specific a chain of posted values. As an 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), our abstraction assumes 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 [IV].

The advantage of our interface is that similar checks and chaining are natural properties to achieve when using blockchain-based consensus systems to instantiate the ledger, since many consensus systems perform the necessary checks as part of their consensus logic. Indeed, we can significantly reduce the cost of deploying our system by using existing ledger systems, including Bitcoin and Ethereum, as they provide these capabilities already, as we discuss in [IV].

We now define our ledger abstraction, which has the following interface:

- Let $H_L$ be a collision-resistant hash function:
  - $\text{Ledger.Post}(\text{Data, CID}) \rightarrow (\text{post, } \sigma)$.

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

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

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

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

The verify algorithm is a public algorithm that will return 1 only if the authenticator $\sigma$ 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.

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 simplicity 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 $\text{Hash}_{i-1}$ and $\text{Hash}_i$ locally.

**Security and finality of the Ledger.** Informally, we require that it is difficult to construct a new pair $(S, \sigma)$ such that $\text{Ledger.Verify}(S, \sigma) = 1$ except as the result of a call to $\text{Ledger.Post}$, even after the adversary has received many authenticator values on chosen strings. We refer to this definition as SUF-AUTH, and it is analogous to the SUF-CMA definition used for signatures. 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. When using “proof of work” ledgers, the authenticators have economic security instead of cryptographic security; we discuss this further in [IV].

### C. Enclave-Ledger Interaction

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

\footnote{We omit the ledger setup algorithm for this description, although many practical instantiations will include some form of setup or key generation.}
D. 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.

Simulation Security. Intuitively, our definition specifies two experiments: a Real experiment in which an adversarial host application runs the real ELI protocol with oracles that implement honest enclave and ledger functionalities respectively, and an Ideal experiment that models the correct and stateful execution of the underlying program $P$ by a trusted party. Our security definition requires that for every p.p.t adversary $\mathcal{H}$ that runs the Real experiment, there must exist an ideal adversary $\mathcal{H}$ that runs the Ideal experiment such that the output of $\mathcal{H}$ is computationally indistinguishable from that of $\mathcal{H}$.

As we discuss in later sections, this intuitive definition may not be strong enough for real deployments. Because in some instantiations (such as proof-of-work blockchains) our ledger may provide only economic security: that is, the cost of forgeries may be impractical. Despite the high cost, in this setting an attacker may be able to forge a small number of ledger proofs. We wish to show that the advantage afforded setting an attacker may be able to forge a small number of forgeries may be impractical. Despite the high cost, in this section we present a specific construction of an Enclave-Ledger Interaction scheme and address its security. Recall that an ELI consists of the three algorithms with the interface described in Figure 2. We present pseudocode for our construction in Algorithms 1, 2 and 3 below.

A. 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 three algorithms with the interface described in Figure 2. We present pseudocode for our construction in Algorithms 1, 2 and 3 below.

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 a pseudorandom function on the hash of the structure returned by the ledger. By the nature of our ledger abstraction, 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
Algorithm 1: Setup

Data: Input: $1^{\lambda}$
Result: Secret $K$ for the enclave and public commitment parameters $pp$

$K \leftarrow \{0, 1\}^{\lambda}$
$pp \leftarrow C\text{Setup}(1^{\lambda})$
Output $(K, pp)$

Algorithm 2: ExecuteApplication

Data: Input: $pp, P$
// Set counter to 0 and state to $\varepsilon$
$S_0 \leftarrow \varepsilon$
$i \leftarrow 0$
$Pub_0 \leftarrow \varepsilon$
// Loop and run the program
while true do
  Obtain $l_i$ from the user
  if $l_i = \perp$ then
    $\perp$ Terminate
  $r_i \leftarrow \{0, 1\}^t$
  $C_i \leftarrow \text{Commit}(pp, (i, l_i, S_i, P); r_i)$
  $(\sigma_i, \text{post}_i) \leftarrow \text{Ledger.Post}((\text{Pub}_i, C_i))$
  $(S_{i+1}, O_i, \text{Pub}_i) \leftarrow \text{ExecuteEnclave}(P, i, S_i, l_i, r_i, \sigma_i, \text{post}_i)$
  Output $(O_i, \text{Pub}_i)$ to the user
  $i \leftarrow i + 1$

Algorithm 3: ExecuteEnclave

Data: Input: $(P, i, S_i, l_i, r_i, \sigma_i, \text{post}_i)$
Internal values: $K, pp$

Result: $(S_{i+1}, O_i, \text{Pub}_i)$ or $\perp$
// Verify and parse the inputs
Assert (Ledger.Verify($\text{post}_i, \sigma_i$))
Assert($\text{post}_i, \text{Hash} = H_L(\text{post}_i, \text{Data}||\text{post}_i, \text{PrevHash})$)
Parse $(\text{Pub}_{i-1}, C_i) \leftarrow \text{post}_i, \text{Data}$
Assert($C_i = \text{Commit}(pp, (i, l_i, S_i, P); r_i)$)
// Compute the $i^{th}$ state encryption key $(k_{i-1}) \leftarrow \text{PRF}_K(\text{post}_i, \text{Hash})$
if $S_i = \varepsilon$ then
  // First execution step, no state.
  Assert($i = 0$)
  $S_i = \varepsilon$
else
  $(S_i, H_P) \leftarrow \text{Decrypt}^{\text{unpad}}(k_i, S_i)$
  Assert($S_i, H_P \neq \perp$)
  Assert($H_P = H(P||i||\text{Pub}_{i-1})$)
  // Compute randomness and $i+1^{th}$ encryption key $(k_{i+1}, \bar{r}_i) \leftarrow \text{PRF}_K(\text{post}_i, \text{Hash})$
  // Run the program and abort if it fails
  $(S_{i+1}, \text{Pub}_i, O_i) \leftarrow P(S_i, l_i, \bar{r}_i)$
  Assert$(S_{i+1}, \text{Pub}_i, O_i) \neq \perp$
  // Encrypt the resulting state
  $S_{i+1} \leftarrow \text{Encrypt}^{\text{pad}}(k_{i+1}, (S_{i+1}, H(P||i+1||\text{Pub}_i))$
  Output $(S_{i+1}, \text{Pub}_i, O_i)$

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 1:** Assuming a secure commitment scheme, a secure authenticated encryption scheme (in the sense of [23]):
that $H$ and $H_L$ are collision resistant; PRF is pseudorandom; and that ledger authentication tags are unforgeable, then the scheme $\Pi = (\text{Setup}, \text{ExecuteEnclave}, \text{ExecuteApplication})$ presented in Algorithms 7, 2, and 3 satisfies Definition 4 (supplied explicitly in the appendix).

We present a proof sketch of Theorem 1 in the appendix. We are limited to a proof sketch because of space constraints. We provide a full proof in the full version of this work.

IV. 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 [11] 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.

A. 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 [6], [31] as well as research systems like Ekiden [21] 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 [6]), 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 that are not assumed to always be trustworthy. In the public setting (e.g., [6]) a single node performs each contract execution, and the remaining nodes simply verify the (deterministic) output of this calculation. This 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 [31] have sought to address this concern by employing trusted execution technology [31]. 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 [19] independently proposed a solution reminiscent of an ELI for securing contracts in this setting. (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. 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)[19]. 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.

### B. Logging and Reporting

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

| Data: Input: $l_i$, $S_i$; Constants: $pk_{auditor}$ |
| Parse (phase, $sk$, $CT$, filename) $\rightarrow S_i$ |
| if $S_i = e$ then // Generate master keypair |
| $(pk, sk) \rightarrow PKKeyGen(1^\lambda)$ |
| $(Pub_{i+1}, sk_{i+1}) \rightarrow (e, pk)$ |
| else if phase = PUBLISH then // Send filename |
| Parse filename $\rightarrow l_i$, $CT_{auditor} \rightarrow PKEnc(pk_{auditor}, filename)$ |
| $(Pub_{i+1}, O_{i+1}) \rightarrow (CT, post)$ |
| else // Decrypt a given file |
| Parse $C \rightarrow l_i$, $(filename, M) \rightarrow PKDec(sk, C)$ |
| if filename $= filename'$ then |
| $(Pub_{i+1} || O_{i+1}) = (e || M)$ |
| else |
| Abort and output ⊥. |
| output $(S_{i+1}, Pub_{i+1}, O_{i+1})$ |

Several cryptographic access control systems require participants to actively log file access patterns to a remote and immutable network location [25]. A popular approach to solving this problem in cryptographic access control systems, leveraged by systems like Hadoop [25], 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[27]. 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[27]. 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 (PKKeyGen, PKEnc, PKDec) and outputs the public key[27]. 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.

#### C. Limited-attempt Password Guessing

Device manufacturers have widely deployed end-to-end file encryption for devices such as mobile phones and cloud backup data[12]. 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[12], trusted enclaves[14], and cloud-based HSMs[42], [41] for backup data. A user authenticates with a relatively weak password such as a PIN and the hardware will release a strong encryption key. To prevent brute force attacks, this stateful hardware must throttle or limit the number of login attempts[14].

Enclave-Ledger Interaction provides an alternative mechanism for limiting the number of guessing attempts on password-based encryption systems. A manufacturer 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 passphrase and the step

---

9This can be accomplished using a broadcast encryption scheme or peer-to-peer key sharing mechanism.

10We 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.

11If 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.

12To provide confidentiality of file accesses, the enclave may encrypt the log entry under the public key of some auditing party.

13Here we require the encryption scheme to be CCA-secure.

14This 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[49], [63].
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.

D. 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. In current ransomware, the centralized system that deliver keys represent a weak point in the ransomware ecosystem. Those systems exposes ransomware operators to tracing. As a result, some operators have fled without delivering key material, as in the famous WannaCry outbreak [37].

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 IV-B.

V. REALIZING THE ENCLAVE AND LEDGER

A. 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 [5], ARM TrustZone [14], or AMD SEV [8]. 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. [48] 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.

Remark. Several recent attacks against trusted co-processors, particularly Intel SGX [62] highlight the possibility that an enclave breach could reveal the key K. These attacks would have catastrophic implications for our protocol. We note that there are several potential mitigations for these attacks. For example, we recommend that an enclave should not directly expose the key K to a given program, but should instead derive a separate key for each program P in case the program contains a vulnerability. Similarly, we emphasize that even in the event of key leakage, industrial systems may be able to renew security through e.g., a microcode update, which will allow the system to derive a new key K from some well-protected internal secret (as Intel did in response to the Foreshadow attack on SGX). Finally, to ensure that a processor...
is using the most recent microcode, the microcode maintainer can list the most recent microcode hash on the ledger and an ELI “bootloader” could use ELI to enforce that the current microcode is up to date. We leave exploration of these ideas to future work.

**Software-Only Options.** A natural software-only equivalent of the enclave is to use pure-software techniques such as virtualization, or cryptographic program obfuscation \[16\]. While software techniques may be capable of hiding secrets from an adversarial user during execution, interactive multi-step obfuscated functionalities are implicitly vulnerable to be run on old state. Unfortunately, there are many negative results in the area of program obfuscation \[16\], and current primitives are not yet practical enough for real-world use \[43\]. However, for specific functionalities this option may be feasible: for example, Choudhuri et al. \[22\] and Jager et al. \[34\] describe protocols based on the related Witness Encryption primitive. New developments in this area could make practical constructions feasible in the next several years.

**B. 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 \[3\], proof of work blockchains like Bitcoin, smart contract systems like Ethereum, and private blockchains.

**Certificate Transparency.** A number of browsers have begun to mandate Certificate Transparency (CT) proofs for TLS certificates \[3\]. 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 programatically submit certificate signing requests, using services like LetsEncrypt, allows us to use CT as a log for any arbitrary data that can be incorporated into an X.509 certificate. 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 \[11\], we require a way to ensure that the 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.

**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 \[2\].

**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 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 forging 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 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 [47], [6]. 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 systems.

Private Blockchains. Many recent systems such as Hyperledger [31] 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 [43]. 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 [11-12].

VI. PROTOTYPE IMPLEMENTATION

To validate our approach we implemented our ELI construction using Intel SGX [5], [45], [35], [10], [15], [53], [52] to implement the enclave and the Bitcoin blockchain to implement the ledger. We embedded a lightweight Javascript engine called Duktape [4] into our enclave, as similar projects implement the ledger. We embedded a lightweight Javascript execution using Intel SGX [5], [45], [35], [10], [15], [53], [100].

Our implementation is only a prototype to demonstrate feasibility and suffers from some limitations, discussed below.

The host application communicates with a local Bitcoin node via RPC to receive blockchain (ledger) fragments for delivery to the enclave, and to send transactions when requested by the enclave. The enclave requires an independent (partial) Bitcoin implementation to verify proof-of-work tags used as ledger authenticators. We based this on the C++ SGX-Bitcoin implementation in the Obscuro project [60].

At startup, the host application loads the Javascript program from a file, initializes the protocol values as in Algorithms [1] and [3] 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 $\text{bitcoin-tx}$ RPC. The first “input” $T.vin[0]$ is set to be an unspent transaction in the local wallet. The first “output” $T.vout[0]$ spends the majority of the input transaction to a new address belonging to the local wallet. The second output $T.vout[1]$ embeds SHA256$$(\text{vin}_i || \text{S}_i || P || \text{CID} || r_i)$$ in an OP_RETURN script. The third output $T.vout[2]$ embeds the public output $\text{Pub}$ 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. The host then sets (1) $\text{post}_i.\text{Data} \leftarrow T.vout$, (2) $\text{post}_i.\text{PrevHash} \leftarrow T.vin[0].\text{Hash}$, (3) $\text{post}_i.\text{CID} \leftarrow \text{chain of transactions from } T \text{ back to the transaction with hash } \text{post}_i.\text{PrevHash}$, (4) $\text{post}_i.\text{Hash} \leftarrow T.\text{Hash}$, and (5) $\sigma_i \leftarrow 6$ blocks confirming $T$.

The host then submits $(\text{post}_i, \sigma_i)$ to the enclave which then performs the following checks: (1) verifies that $\sigma_i$ is valid and has sufficiently high block difficulty (2) the blocks in $\sigma_i$ are consecutive (3) $T.vout[0], T.vout[1]$ embed the correct data and (4) the transactions in $\text{post}_i.\text{CID}$ 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 C-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 SHA256. One note is that instead of hashing all of CID into the ciphertext, we include only $\text{post}_0.\text{PrevHash}$, which keeps CID constant throughout the rounds.

Implementation Limitations. We chose to use Intel SGX to implement our enclave because it is a widely accepted, secure execution environment. However, SGX is significantly more powerful than the enclaves we model, including access to trusted time and monotonic counters. Although we use SGX, we do not leverage any of these additional features to make sure our implementation matches our model. Our Bitcoin implementation of the ledger is slow and would likely not be suitable for production release. Finally, we implement our applications in Javascript so the Javascript virtual machine will insulate the enclave code from host tampering. The use of Javascript, however, could lead to other complications.

Measurements. To avoid spending significant money on the Bitcoin main network, we tested our implementation on a private regression regtest. This also allows us to control the rate at which blocks are mined. The most time-consuming portion of an implementation using the mainnet or 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).
Our experiments used a simple 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 3 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 3 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.

VII. 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-ledge 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


APPENDIX

Proof Sketch. Due to lack of space, we provide a proof sketch with many of the details omitted. A complete proof of security is available in the full version of this work.

We give a simulation based definition for ELI, and we use two basic experiments in our proof. In the Real experiment, we consider an interaction in which an adversarial host \( H \) interacts with an honest Ledger Oracle and an honest Enclave Oracle, as described in [31] to execute the ELI protocol. The Ideal experiment has adversarial ideal host \( H' \) that interacts with a trusted, multi-step, computational functionality, exposed as a Compute Oracle. At each step of the experiment, this functionality takes as input a program, a program input, and a “session ID” provided by \( H' \), 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.

We augment these experiments slightly to account for ledger authenticator forgability. Particularly when the ledger authenticator tags are secured economically, we must ensure that the security degrades gracefully in face of a small number of forgeries. While we cannot prevent an attack from obtaining some advantage from successfully forging an authenticator, we can successfully bound the attacker’s capability in this setting. To capture this notion in our proof, we give the Real experiment adversary access to a Forgery Oracle. In the
Ideal experiment, we provide a Fork Oracle that allows the adversary to run a single step of computation using an older state of their choosing. In each experiment, the adversary can only make a maximum of $q_{\text{forge}}$ queries to these oracles.

We prove that for every p.p.t. adversarial real-world hosts $\mathcal{H}$, there must exist a p.p.t. ideal-world host $\mathcal{H}^*$ that does “as well” in the ideal experiment as the real host does in the real experiment. Note that this would imply that a real-world adversary with the ability to forge $q_{\text{forge}}$ authenticators would be able to make exactly $q_{\text{forge}}$ single step computations, a very reasonable bound. Formally:

**Definition 1 (Simulation security for ELI):** An ELI scheme $\Pi = (\text{ExecuteApplication}, \text{ExecuteEnclave})$ is simulation-secure if for every p.p.t. adversary $\mathcal{H}$, sufficiently large $\lambda$, and non-negative $q_{\text{forge}}$, there exists a p.p.t. $\mathcal{H}^*$ such that the following holds:

$$\text{Real}(\mathcal{H}, \lambda, q_{\text{forge}}) \approx \text{Ideal}(\mathcal{H}, \lambda, q_{\text{forge}})$$

**Proof Sketch:** Given the space concerns, we cannot give the full description of the ideal-world adversary $\mathcal{H}^*$ Intuitively, $\mathcal{H}^*$ mediates the communication between $\mathcal{H}$ and the oracles. When $\mathcal{H}$ attempts to query the Ledger Oracle, $\mathcal{H}^*$ forwards it to the real ledger. When $\mathcal{H}$ queries the Enclave Oracle, $\mathcal{H}^*$ references its internal records of previous interactions and decides if it is (1) an invalid query, (2) a previous query, (3) a query with a forged authenticator, or (4) a fresh query. In case (1), $\mathcal{H}$ aborts. In case (2), $\mathcal{H}^*$ replays the appropriate old output. In case (2), $\mathcal{H}^*$ forwards the request to the Forgery Oracle. And finally, in case (3), $\mathcal{H}^*$ forwards the query to the Compute Oracle. The bulk of the proof is showing that $\mathcal{H}$ can distinguish each of these cases successfully.

**Discussion.** Let $D$ be a p.p.t. distinguisher that succeeds in distinguishing $\mathcal{H}^*$’s output in the Ideal experiment from $\mathcal{H}^*$’s output in the Real experiment with non-negligible advantage. The proof proceeds via a series of hybrids, where in each hybrid $\mathcal{H}$ interacts as in the Real experiment with non-negligible advantage. The proof concludes by invoking the hybrid lemma to show that the Real experiment and Ideal experiment are indistinguishable.

**Game 0** is simply the Real experiment. We now sketch the remaining games:

**Game 1 (Abort on [adversary]-forged authenticators.)** If $\mathcal{H}$ queries the Enclave oracle on $(post, \sigma)$ such that (1) Ledger.Verify$(post, \sigma) = 1$, 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 $\text{Event}_{\text{forge}}$. This event occurs with at most negligible probability if authenticators are unforgeable.

**Game 2 (Abort on hash collisions.)** If $\mathcal{H}$ causes the functions $H_l, H_u$ to be evaluated on inputs $s_1 \neq s_2$ such that $H_l(s_1) = H_l(s_2)$ or $H_u(s_1) = H_u(s_2)$, then abort and output $\text{Event}_{\text{hashcoll}}$. This occurs with at most negligible probability if the hash is collision resistant.

**Game 3 (Abort on commitment collisions.)** If $\mathcal{H}$ queries the Enclave oracle at steps $i,j$ where $C_i = C_j = \text{Commit}(pp, (i, 1, \hat{S}, P, post, \text{CID}); r_i)$ and yet $(i, l, \hat{S}, P, post, \text{CID}) \neq (j, l, \hat{S}, P, post, \text{CID})$, then abort and output $\text{Event}_{\text{binding}}$. This occurs with at most negligible probability if the commitment is binding.

**Game 4 (Duplicate Enclave calls give identical outputs.)** If $\mathcal{H}$ queries the Enclave oracle repeatedly on the same values $(P_i, i, l, \hat{S}_i, post_i)$ (here we exclude $\sigma_i$) and the oracle (as implemented in the previous hybrid) does not output $\bot$, replace the response to all repeated queries subsequent to the first query with the same result as the first query. This does not change the distribution because the scheme is deterministic.

**Game 5 (Abort on colliding ledger hashes.)** If $\mathcal{H}$ queries the Enclave oracle on two distinct inputs $(P_i, i, l, \hat{S}_i, post_i, \sigma_i)$ and $(P_j, j, l, \hat{S}_j, post_j, \sigma_j)$, and if the two inputs do not represent repeated inputs (according to Game 4), then: if both $(post_i, \sigma_i)$ and $(post_j, \sigma_j)$ are valid outputs of the Ledger oracle and yet post$_i$.Hash = post$_j$.Hash then abort and output $\text{Event}_{\text{ledgercoll}}$. This occurs with at most negligible probability if the hash is collision resistant.

**Game 6 (\(\mathcal{H}\) 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 $\bot$, (2) the inputs to the two calls are not identical (this would be excluded by the earlier hybrids), and (3) the pair $(post_i, \sigma_i)$ are in the Ledger table, and (4) post$_i$.Hash matches two distinct entries in the Ledger table, then abort and output $\text{Event}_{\text{ledgerrepeat}}$. This event occurs with at most negligible probability.

**Game 7 (Replace the session keys and pseudorandom coins with random strings.)** If Enclave does not abort or is not called on repeated inputs, then the pair $(k_{i+1}, r_i)$ is sampled uniformly at random and recorded in a table for later use. The use of a PRF makes this hybrid indistinguishable from the previous.

**Game 8 (Reject inauthentic ciphertexts.)** If $\mathcal{H}$ queries the Enclave oracle on an input $S_i \neq \varepsilon$ such that (1) the oracle does not reject the input, (2) Decrypt$(k_{i}, S_i)$ does not output $\bot$, and yet (3) the pair $(S_i, k_i)$ was not generated during a previous query to Enclave, then abort and output $\text{Event}_{\text{auth}}$. If the scheme is AE, this event occurs with at most negligible probability.

**Game 9 (Abort if inputs are inconsistent.)** On $\mathcal{H}$’s the $i^{th}$ query to the Enclave oracle, when the input $S_i \neq \varepsilon$, let $(post\text{CID}', P', i', \text{Pub}_{i'})$ be the inputs/outputs associated with the previous Enclave call that produced $S_i$. 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 $(post\text{CID}, P, i - 1, \text{Pub}_{i-1}) \neq (post\text{CID}', P', i', \text{Pub}_{i'-1})$ differ from those associated with the previous call to the Enclave, then abort and output $\text{Event}_{\text{mismatch}}$. This event cannot occur in the current protocol.

**Game 10 (Replace ciphertexts with dummy ciphertexts.)** This hybrid modifies the previous as follows: we modify the generation of each ciphertext $S_{\text{out}}$ to encrypt the unary string $(\ell, 1^\ell)$. If the scheme is AE, then this hybrid cannot be distinguished from the previous.