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], [1], or 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 [5], [14], [8]. 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].[1] However, such countermeasures

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
http://dx.doi.org/10.14722/ndss.2019.23xxx
www.ndss-symposium.org

[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 [58], [39], [57] using relatively inexpensive physical and electronic attacks.

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 [48], [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 devlopers be alerted on the ledger that user activity is anomolous, 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 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.

Previous and concurrent work. A manuscript of this work was initially published in February 2017 [38]. Several previous and concurrent works haved 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

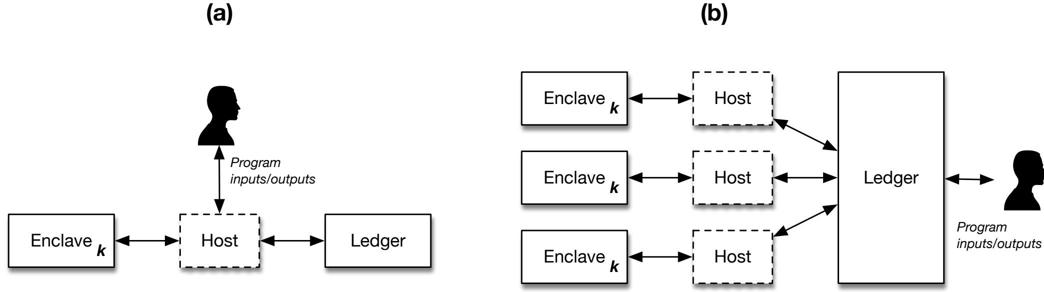

Fig. 1. Two example ELI deployments. In the basic scenario (a) a single TEE (with a hard-coded secret key K) interacts with a ledger functionality via a (possibly adversarial) host application. Program inputs are provided by a user via the host machine. In scenario (b) multiple copies of the same enclave running on different host machines interact with the ledger (*e.g.,* as in a private smart contract system), 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.

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:

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

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.[2] 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.

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

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.[3]

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 σ, 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 to this encrypted state an iteration counter i which identifies the next step of the program to be executed.

To execute the ith invocation of the program, the host first commits its next program input I_i to the ledger. This can be

[3]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.

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).[4] It verifies the following conditions:

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, 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.[5]

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. 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 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

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

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

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 §IV.

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 migrates 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 [18], [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 $\mathcal{E}$, 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.

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

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

When a user inputs 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}.

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 identified by a session identifier, which we denote by CID. Thus, the main difference between a one-time and many-time

[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 [13].

[7] See [57] for an example of how such systems can be defeated when state is recorded in standard NAND hardware, rather than full tamper-resistant hardware.

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 §IV.

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 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 naïve 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 §V-B.

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 §V-B.

We now define our ledger abstraction, which has the following interface.[8] 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 (post, σ).
- $\mathsf{Ledger.Verify}(\mathsf{post}, \sigma) \rightarrow \{0, 1\}$.
  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.

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 Hash_{i-1} and Hash_i locally.

Security and finality 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. We refer to this definition as $\mathsf{SUF\text{-}AUTH}$, and it is analogous to the $\mathsf{SUF\text{-}CMA}$ 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 §V-B.

C. Enclave-Ledger Interaction

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

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

$\mathsf{Setup}(1^\lambda) \rightarrow (K, \mathsf{pp})$. 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.

$\mathsf{ExecuteEnclave}_{K,\mathsf{pp}}((P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \sigma_i, \mathsf{post}_i)) \rightarrow (\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_i)$. This algorithm is run by the enclave, which is configured with K, pp. At the ith 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.

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

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 $\hat{\mathcal{H}}$ that runs the **Ideal** experiment such that the output of $\mathcal{H}$ is computationally indistinguishable from that of $\hat{\mathcal{H}}$.

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 such an attacker is *minimized*. In the appendix we strengthen our definition to allow the attacker to forge a limited number of ledger authentication tags.

III. 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. Let λ be a security parameter. Let ℓ be a non-negative integer where $\ell = \mathrm{poly}(\lambda)$. In our constructions we define $\mathsf{Verify}(\cdot)$ as a primitive that verifies a statement, and aborts the program (with output $\perp$) 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.

Deterministic authenticated encryption. We require a symmetric authenticated encryption scheme consisting of the algorithms $(\mathsf{Encrypt}, \mathsf{Decrypt})$ 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 $\mathsf{Encrypt}$ 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 define the specialized algorithm $\mathsf{Encrypt}^{\mathsf{pad}}$ as one that *pads* the plaintext to (a maximum state size) n bits prior to encrypting it, and define $\mathsf{Decrypt}^{\mathsf{unpad}}$ as removing this padding.

Pseudorandom Functions. Our construction uses a pseudorandom function family $\mathsf{PRF} : \{0,1\}^\ell \times \{0,1\}^* \rightarrow \{0,1\}^{2\ell}$ that on input an ℓ-bit key and a string of arbitrary length, 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.,* [55]).

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^λ
Result: Secret K for the enclave and public
        commitment parameters pp
K ←$ {0,1}^λ
pp ← CSetup(1^λ)
Output (K, pp)
```

Algorithm 2: ExecuteApplication

```
Data: Input: pp, P
// Set counter to 0 and state to ε
S_0 ← ε
i ← 0
Pub_0 ← ε
// Loop and run the program
while true do
    Obtain I_i from the user
    if I_i = ⊥ then
        Terminate
    r_i ←$ {0,1}^ℓ
    C_i ← Commit(pp, (i, I_i, S_i, P); r_i)
    (σ_i, post_i) ← Ledger.Post ((Pub_i, C_i))
    (S_{i+1}, O_i, Pub_i) ← ExecuteEnclave(P, i, S_i, I_i, r_i,
        σ_i, post_i)
    Output (O_i, Pub_i) to the user
    i ← i + 1
```

Algorithm 3: ExecuteEnclave

```
Data: Input: (P, i, S_i, I_i, r_i, σ_i, post_i)
      Internal values: K, pp
Result: (S_{i+1}, O_i, Pub_i) or ⊥
// Verify and parse the inputs
Assert (Ledger.Verify(post_i, σ_i))
Assert(post_i.Hash = H_L(post_i.Data‖post_i.PrevHash))
Parse (Pub_{i−1}, C_i) ← post.Data
Assert(C_i = Commit(pp, (i, I_i, S_i, P); r_i))
// Compute the i^th state encryption key
(k_i, ·) ← PRF_K(post_i.PrevHash)
if S_i = ε then
    // First execution step, no state.
    Assert(i = 0)
    S_i = ε
else
    (S_i, H_P) ← Decrypt^unpad(k_i, S_i)
    Assert((S_i, H_P) ≠ ⊥)
    Assert(H_P = H(P‖i‖Pub_{i−1}))
// Compute randomness and i+1^th
   encryption key
(k_{i+1}, r̄_i) ← PRF_K(post_i.Hash)
// Run the program and abort if it
   fails
(S_{i+1}, Pub_i, O_i) ← P(S_i, I_i; r̄_i)
Assert((S_{i+1}, Pub_i, O_i) ≠ ⊥)
// Encrypt the resulting state
S_{i+1} ← Encrypt^pad(k_{i+1}, (S_{i+1}, H(P‖i+1‖Pub_i))
Output (S_{i+1}, 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 [54]); that H *and* $\mathsf{H_L}$ *are collision resistant;* PRF *is pseudorandom; and that ledger authentication tags are unforgeable, then the scheme* $\Pi = (\mathsf{Setup}, \mathsf{ExecuteEnclave}, \mathsf{ExecuteApplication})$ *presented in Algorithms 1, 2 and 3 satisfies Definition 1 (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 §III 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 who 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 [33] 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.[9] 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).[10] 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: I_i , S_i; Constants: pk_auditor
Parse (phase, sk, CT, filename) ← S_i
if S_i = ε then // Generate master keypair
    (pk, sk) ← PKKeyGen(1^λ)
    S_{i+1} = (PUBLISH, sk, ·, ·)
    (Pub_{i+1}, O_{i+1}) ← (ε, pk)
else if phase = PUBLISH then // Send filename
    Parse filename ← I_i
    CT_auditor ← PKEnc(pk_auditor, filename)
    S_{i+1} ← (DECRYPT, sk, CT, I_i)
    (Pub_{i+1}, O_{i+1}) ← (CT, post)
else // Decrypt a given file
    Parse C ← I_i
    (filename', M) ← PKDec(sk, C)
    if filename = filename' then
        S_{i+1} = (PUBLISH, sk, ·, ·)
        (Pub_{i+1} || O_{i+1}) = (ε || 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.[11] 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.[12] 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.[13] 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], [11]. 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 passcode 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 passcode *and* the step

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

[10]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.

[11]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.

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

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

[14]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[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 [64]. In current ransomware, the centralized system that deliver keys represent a weak point in the ransomware ecosystem. Those systems exposes ransomware operators to tracing [59]. As a result, some operators have fled without delivering key material, as in the famous WannaCry outbreak [37].

Algorithm 5: Ransomware $P_{ransomware}$

```
Data: Input: I_i , S_i; Randomness r_i;
Parse (K, R, pk) ← S_i
if S_i = ε then // Generate Key, Set Ransom
    Parse (R, pk) ← I_i
    K ← KDF(r_i)
    output S_{i+1} ← (K, R, pk)
else // Release Key on Payment
    Parse (t, σ) ← I_i
    if (BlockchainVerify(t, σ) = 1) then
        if (t.amount > R and t.target = pk) then
            output O_i = K
    output O_i = ⊥
```

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[43].

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 being 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 programmatically 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 §II-B, 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 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 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

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%

Fig. 3. 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 a local Bitcoin regtest network

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 system.

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 [33]. 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 §II-B.

VI. Prototype Implementation

To validate our approach we implemented our ELI construction using Intel SGX [5], [45], [35], [10], [15], [53], [32]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 have done in the past [46]. Source code can be found at https://github.com/JHU-ARC/state_for_the_stateless/. 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 sends transaction 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, 2 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 `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.[15] The host then sets (1) $\mathsf{post}_i.\mathsf{Data} \leftarrow T.\mathsf{vout}$, (2) $\mathsf{post}_i.\mathsf{PrevHash} \leftarrow T.\mathsf{vin}[0].Hash$, (3) $\mathsf{post}_i.\mathsf{CID} \leftarrow$ chain of transactions from T back to the transaction with hash $\mathsf{post}_0.\mathsf{PrevHash}$, (4) $\mathsf{post}_i.\mathsf{Hash} \leftarrow T.Hash$, and (5) $\sigma_i \leftarrow$ 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 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 $\mathsf{post}_0.\mathsf{PrevHash}$, 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).

[15] 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.

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-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

[1] "Namecoin," https://namecoin.org/, November 2016. [Online]. Available: https://namecoin.org/

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

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

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

[5] "Intel Software Guard Extensions (Intel SGX)," https://software.intel.com/en-us/sgx, 2018.

[6] "The Ethereum Project," https://www.ethereum.org/, 2018.

[7] "TPM Reset Attack," Available at http://www.cs.dartmouth.edu/~pkilab/sparks/, 2018.

[8] Advanced Microchip Devices, Available at https://developer.amd.com/sev/, 2018.

[9] Amazon, "AWS Step Functions," Available at https://aws.amazon.com/step-functions/, 2018.

[10] I. Anati, S. Gueron, S. Johnson, and V. 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*, vol. 13, 2013.

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

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

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

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

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

[16] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," Cryptology ePrint Archive, Report 2001/069, 2001, http://eprint.iacr.org/2001/069.

[17] M. Bellare, M. Fischlin, S. Goldwasser, and S. Micali, "Identification protocols secure against reset attacks," in *EUROCRYPT '01*, B. Pfitzmann, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 495–511. [Online]. Available: https://doi.org/10.1007/3-540-44987-6_30

[18] J. Bonneau, "The science of guessing: Analyzing an anonymized corpus of 70 million passwords," in *IEEE S&P (Oakland) '12*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 538–552. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.49

[19] M. Bowman, A. Miele, M. Steiner, and B. Vavala, "Private Data Objects: an Overview," *ArXiv e-prints*, Jul. 2018.

[20] R. Canetti, O. Goldreich, S. Goldwasser, and S. Micali, "Resettable zero-knowledge (extended abstract)," in *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, ser. STOC '00. New York, NY, USA: ACM, 2000, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/335305.335334

[21] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *CoRR*, vol. abs/1804.05141, 2018. [Online]. Available: http://arxiv.org/abs/1804.05141

[22] A. R. Choudhuri, M. Green, A. Jain, G. Kaptchuk, and I. Miers, "Fairness in an unfair world: Fair multiparty computation from public bulletin boards," in *CCS '17*, 2017, https://eprint.iacr.org/2017/1091.

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

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

[25] A. Foundation, "Hadoop Key Management Server (KMS) - Documentation Sets," Available at https://hadoop.apache.org/docs/stable/hadoop-kms/index.html, 2018.

[26] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," Cryptology ePrint Archive, Report 2013/451, 2013, http://eprint.iacr.org/2013/451.

[27] B. Giller, "Implementing Practical Electrical Glitching Attacks," in *BlackHat '15*, 2015.

[28] Google Inc., "Google Cloud Functions," Available at https://cloud.google.com/functions/, 2018.

[29] R. Goyal and V. Goyal, "Overcoming cryptographic impossibility results using blockchains," Cryptology ePrint Archive, Report 2017/935, 2017, https://eprint.iacr.org/2017/935.

[30] Handshake, "Handshake protocol," Available at https://handshake.org/, 2018.

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

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

[33] ——, "Hyperledger Sawtooth," Available at http://hyperledger.org/projects/sawtooth, 2018.

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

[35] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel Software Guard Extensions: EPID Provisioning and Attestation Services," 2016.

[36] A. Juels, A. Kosba, and E. 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 283–295. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978362

[37] M. Kan, "Paying the WannaCry ransom will probably get you nothing. Here's why." *PCWorld*, 2017, available at https://www.pcworld.com/article/3196880/security/paying-the-wannacry-ransom-will-probably-get-you-nothing-heres-why.html.

[38] G. Kaptchuk, I. Miers, and M. 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.

[39] B. Kauer, "OSLO: Improving the Security of Trusted Computing," in *Usenix '07*. Berkeley, CA, USA: USENIX Association, 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362919

[40] 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)*, May 2016, pp. 839–858.

[41] I. 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.

[42] 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.

[43] K. Lewi, A. J. Malozemoff, D. Apon, B. Carmer, A. Foltzer, D. Wagner, D. W. Archer, D. Boneh, J. Katz, and M. 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*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 981–992. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978314

[44] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: Rollback protection for trusted execution," Cryptology ePrint Archive, Report 2017/048, 2017, http://eprint.iacr.org/2017/048.

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[46] M. Milutinovic, W. He, H. Wu, and M. Kanwal, "Proof of luck: an efficient blockchain consensus protocol," Cryptology ePrint Archive, Report 2017/249, 2017, https://eprint.iacr.org/2017/249.

[47] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system, 2008," 2008. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[48] K. Nayak, C. W. Fletcher, L. Ren, N. Chandran, S. Lokam, E. Shi, and V. Goyal, "HOP: hardware makes obfuscation practical," in *NDSS '17*, 2017.

[49] D. Paletta, "FBI Chief Punches Back on Encryption," *Wall Street Journal*, July 2015. [Online]. Available: http://www.wsj.com/articles/fbi-chief-punches-back-on-encryption-1436217665

[50] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 379–394. [Online]. Available: https://doi.org/10.1109/SP.2011.38

[51] K. Poulsen, "DirecTV attacks hacked smart cards," *The Register*, 2001, https://www.theregister.co.uk/2001/01/25/directv_attacks_hacked_smart_cards/.

[52] A. Project, "Full-Disk Encryption," Available at https://source.android.com/security/encryption/full-disk.html, 2017.

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

[54] P. Rogaway, "Authenticated encryption with associated data," in *CCS '02*. ACM Press, 2002.

[55] ——, "Formalizing human ignorance," in *VIETCRYPT 2006*, P. Q. Nguyen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 211–228.

[56] D. 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.

[57] S. Skorobogatov, "The bumpy road towards iPhone 5c NAND mirroring," *CoRR*, vol. abs/1609.04327, 2016. [Online]. Available: http://arxiv.org/abs/1609.04327

[58] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *CHES '02*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_2

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

[60] M. Tran, L. Luu, M. S. Kang, I. Bentov, and P. Saxena, "Obscuro: A bitcoin mixer using trusted execution environments," Cryptology ePrint Archive, Report 2017/974, 2017, http://eprint.iacr.org/2017/974.

[61] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay, "Measuring real-world accuracies and biases in modeling password guessability," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 463–481. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ur

[62] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [**?**].

[63] N. Weaver, "iPhones, the FBI, and Going Dark," *Lawfare Blog*, August 2015. [Online]. Available: https://www.lawfareblog.com/iphones-fbi-and-going-dark

[64] K. 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.

[65] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 270–282. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978326

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 user $\mathcal{H}$ interacts with an honest **Ledger Oracle** and an honest **Enclave Oracle**, as described in §III, to execute the ELI protocol. The **Ideal** experiment has adversarial *ideal* host $\hat{\mathcal{H}}$ 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 $\hat{\mathcal{H}}$, 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.

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_{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 $\hat{\mathcal{H}}$ 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_{forge} authenticators would be able to make exactly q_{forge} single step computations, a very reasonable bound. Formally:

Definition 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:*

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

Proof Sketch: Given the space concerns, we cannot give the full description of the ideal-world adversary $\hat{\mathcal{H}}$. Intuitively, $\hat{\mathcal{H}}$ mediates the communication between $\mathcal{H}$ and the oracles. When $\mathcal{H}$ attempts to query the **Ledger Oracle**, $\hat{\mathcal{H}}$ forwards it to the real ledger. When $\mathcal{H}$ queries **Enclave Oracle**, $\hat{\mathcal{H}}$ 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), $\hat{\mathcal{H}}$ aborts. In case (2), $\hat{\mathcal{H}}$ replays the appropriate old output. In case (2), $\hat{\mathcal{H}}$ forwards the request to the **Forgery Oracle**. And finally, in case (3), $\hat{\mathcal{H}}$ forwards the query to the **Compute Oracle**. The bulk of the proof is showing that $\hat{\mathcal{H}}$ can distinguish each of these cases successfully.

Discussion. 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. 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. We enumerate the hybrids below but, due to space constraints, omit the proof that each pair is computationally indistinguishable. 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, 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 $\mathsf{Event}_{\mathsf{forge}}$. This event occurs with at most negligible probability if authenticators are unforgeable.

Game 2 (**Abort on hash collisions.**) If $\mathcal{H}$ 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 $\mathsf{Event}_{\mathsf{hashcoll}}$. 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 $C_i = C_j = \mathsf{Commit}(\mathsf{pp}, (i, \mathsf{I}_i, \mathcal{S}_i, P, \mathsf{post}_i.\mathsf{CID}); r_i) = \mathsf{Commit}(\mathsf{pp}, j\|\mathsf{I}_j\|\mathcal{S}_j\|\mathsf{post}_i.\mathsf{CID}; r_j)$ 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 $\mathsf{Event}_{\mathsf{binding}}$. 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 $(P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$ (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. This does not change the distribution because the scheme is deterministic.

Game 5 (**Abort on colliding ledger hashes.**) If $\mathcal{H}$ quries 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 $\mathsf{Event}_{\mathsf{ledgercoll}}$. This occurs with at most negligible probability if the hash is collision resistant.

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 $\mathsf{Event}_{\mathsf{ledgerrepeat}}$. 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 $(k_{i+1}, \bar{r}_i)$ 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 $\mathcal{S}_i \neq \varepsilon$ 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 $\mathsf{Event}_{\mathsf{auth}}$. 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 ith 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 $\mathsf{Event}_{\mathsf{mismatch}}$. 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 $\mathcal{S}'_{out}$ to encrypt the unary string $(1^{\mathsf{Max}(P)}, 1^{\ell})$. If the scheme is AE, then this hybrid cannot be distinguished vrom the previous.