Managing Secrets with Consensus Networks: Fairness, Ransomware and Access Control

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

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

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

Abstract

In this work we investigate the problem of using public consensus networks – exemplified by systems like Ethereum and Bitcoin – to perform cryptographic functionalities that involve the manipulation of *secret* data, such as cryptographic access control. We consider a hybrid paradigm in which a secure client-side functionality manages cryptographic secrets, while an online consensus network performs public computation. Using this approach, 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) from a *stateless* client-side functionality, which can be implemented using inexpensive trusted hardware or even purely cryptographic functionalities such as Witness Encryption. We then describe a number of practical applications that can be achieved today. These include rate limited mandatory logging; strong encrypted backups from weak passwords; enforcing fairness in multi-party computation; and destructive applications such as *autonomous ransomware*, which allows for payments without an online party.

1 Introduction

Cryptography is a powerful tool for managing access to protected data. From disk encryption [Bla94, Bla93, The] to modern database encryption [PRZB11], cryptography is often used to restrict which data objects can be accessed by users. Many modern access control systems perform *dynamic*, or interactive cryptographic access control: employing a trusted server or a local tamper-resistant module to store and selectively deliver cryptographic keys to authorized clients when various conditions are met [Krs16, KRS$^+$03].

These solutions have many advantages: they can enforce centralized logging and revocation, and can provide assurances such as restricting the user to a limited of authentication attempts [App16, Sko16, Krs16, Pro]. Unfortunately, such dynamic access control systems have corresponding disadvantages: in the online case, they require the construction of expensive centralized infrastructure that is vulnerable to remote attacks and DoS. In the purely offline case (*e.g.,* using trusted local hardware) it is challenging to enforce guarantees such as mandatory file access logging, or to revoke stolen devices. Moreover, sophisticated attackers may be able to "rewind" the non-volatile state of some inexpensive devices [Sko16], disabling security guarantees such as passcode guessing limits.

In recent years a new class of distributed system has evolved. Loosely categorized as *consensus networks* and exemplified by Bitcoin and Ethereum [Nak12, etha], these systems employ highly decentralized and volunteer-run networks to evaluate public rules that manage access to various resources. The most famous of these systems, Bitcoin [Nak12], uses consensus to enforce monetary payment rules. However, more recent systems have been proposed for applications ranging from identity management [nam16, GGM13] to resource management in distributed systems [JMH15, GRFJ14, BP15]. The benefits of using consensus networks are twofold: (1) these volunteer-run networks are readily available and thus do not require the deployment of expensive new infrastructure, and (2) compromising such networks often requires the attacker to deploy significant resources, computational or otherwise.

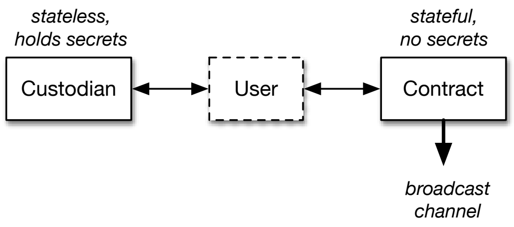

Figure 1: Custodian-Contract interaction. The Custodian interacts with a Contract functionality via a (possibly adversarial) user. The Contract holds state and has access to a public input/output channel (*e.g.,* an append-only blockchain ledger), while the Custodian is a local trusted functionality that holds secret data.

While public consensus networks have evolved to allow the execution of arbitrary user-defined programs (called "smart contracts") [etha], these networks have a critical limitation that restricts their usefulness for certain interactive cryptographic protocols. Namely, public consensus systems *do not provide the ability to securely store and operate on secret data* such as cryptographic keys. While hypothetically it might be possible to upgrade these systems to incorporate secret storage, in practice this is challenging. It would require either a large amount of trust to be placed in a small number of reliable nodes, or else it would necessitate new and complex multiparty computation and proactive secret sharing protocols to distribute and verify trust across the larger (and constantly churning) volunteer network. This challenge greatly limits the usefulness of these networks in controlling access to certain types of encrypted data.

Our contributions. In this work we take a step towards addressing the management of secret data using public consensus networks. The main building block in our proposals is a new paradigm we refer to *Custodian-Contract interaction.* This proposal divides each interactive secret computation into a protocol run between two parties: a stateless, client-side functionality called the *Custodian* that securely manages secrets, and a *Contract* that enforces consensus rules in a public network. The Custodian may be implemented using tamper-resistant hardware, or cryptographic obfuscation techniques. The *Contract* may be implemented using a public online consensus network. The communication between each functionality is facilitated by a possibly adversarial *User*, as shown in Figure 1.

We observe that this idea has some powerful implications. First, by instantiating the Contract on a computationally-supported consensus network (*e.g.,* a smart contract system that uses proof-of-work blockchains to achieve consensus), we immediately obtain a means for the Custodian to cryptographically validate the authenticity of decisions made by the consensus network. In this setting, the Custodian functionality can, using only modest computational effort, verify a fragment of the blockchain to ensure (with reasonable probability) that it is a legitimate consensus output.[1] Under the economic assumption that the consensus network possesses significantly more computational power than any likely adversary, this ensures that interactions between the Contract and Custodian are authenticated.[2] This holds even when the adversary (*i.e.,* the User) provides the communication channel between the two components.

Second, we observe that the combination of Contract and Custodian can achieve properties that may not be achievable using trusted hardware alone. Interaction with a Contract provides the ability to condition certain secret calculations on *public actions* taken on the consensus network. In particular, this allows us to condition decryption operations on the publication of certain messages, including monetary payments made to specific parties on the network. This enables practical applications such as enforced file access logging, device revocation, and even the enforcement of payments in exchange for delivery of secrets. 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. Finally, we show that these techniques can be used to facilitate *enforced*

[1]A similar idea was proposed for implementing "sidechains" in Bitcoin [BCD$^+$14], and is related to Simplified Payment Verification in Bitcoin clients.

[2]As an example of the viability of this assumption, we note that the current Bitcoin network executes approximately 2^{70} SHA2 hashes every 10 minutes.

completion of multiparty computation (MPC), which provides a new solution to an well-known problem related to fairness of MPC protocols [Cle86].

As a practical matter, we show that – given access to an appropriate smart Contract network – an extremely lightweight and *stateless* Custodian (such as the hardware proposed in [NFR$^+$17]) can now be used to perform interactive and stateful calculations involving secret data. We show that in practice this simple enhancement enables us to construct complex and access control policies that depend on state, such as limiting the number of login attempts to a password-encrypted device – even when an attacker is able to modify and rewind the contents of the Custodian's local storage [Sko16]. This capability is particularly exciting when the Custodian is implemented using purely software techniques, such as cryptographic obfuscation [BCP14, BR14] or inexpensive stateless hardware co-processors [NFR$^+$17, DMMQN11].

1.1 Applications

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

Limiting password guessing. Many cryptographic access control systems employ passwords to control access to encrypted filesystems [App16, Pro] and cloud backup images (*e.g.,* Apple's iCloud Keychain [Krs16]). This creates a tension between the requirement to support easily memorable passwords (such as device PINs), while simultaneously preventing attackers from simply *guessing* users' relatively weak passwords [Bon12, USB$^+$15].[3] One approach to addressing this is to incorporate tamper-resistant hardware such as onboard co-processors [App16, Pro, ARM] and Hardware Security Modules [Krs16]. However, these systems are expensive (particularly in the online backup case [Krs16]) and may fail catastrophically when an attacker can rewind state.[4] We show that using our paradigm we can safely enforce *passcode guessing limits* even when using inexpensive hardware that cannot guarantee immutable state [Sko16].

Fairness in multi-party computation Multi-party computation (MPC) is a powerful cryptographic primitive that allows many players to cooperatively compute arbitrary functionality without revealing their individual inputs. A well known limitation of MPC protocols is that of ensuring *fairness.* When at least half the players in a computation are malicious, an attacker may be able to abort the computation before the honest parties learn the output [Cle86]. As a novel application of our technique, we construct a new primitive called *fair encryption*, that ensures a ciphertext can be decrypted *if and only if it has been published on a blockchain.* Using this primitive we show how to construct a new class of fair MPC protocols in which the malicious party provably learns no output unless all parties learn the output of the protocol. This advances a line of work on using blockchain-based protocols to *penalize* misbehaving parties in MPC protocols [GK10, ADMM14].

Autonomous ransomware. Ransomware is a class of malware that encrypts a victim's files, then demands a monetary ransom in exchange for decryption. Modern ransomware platforms are tightly integrated with cryptocurrencies such as Bitcoin, which act as both the ransom currency and a communication channel to the attacker [Sin16]. Once a system has been infected, users must transmit a an encrypted key package along with a ransom payment to the attacker, who responds with the necessary decryption keys. The need to deliver secret keys is a fortunate weakness in the current ransomware paradigm, as it is both costly and increases the probability that she will be traced by law enforcement [Goo13]. More critically, this makes ransom payment a risky proposition for the victim, who cannot verify *a priori* that paying the ransom will restore access to data.[5] In this work we remark on a potentially

[3]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 [App].

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

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

destructive application of the Custodian-Contract paradigm: the creation of ransomware that operates autonomously – from infection to decryption – with no need for remote parties to deliver secret keys. This ransomware employs local trusted hardware or obfuscation to store a decryption key for a user's data, and conditions decryption of a user's software on payments made on a public consensus network.

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

1.2 Intuition

We now briefly present the intuition behind our construction. Recall that the goal is to perform interactive computation on secret data, using a *stateless* but trustworthy computing environment (the Custodian) in combination with a stateful smart contract. For some secret data s, we will model the desired computation as a probabilistic reactive functionality of the form:

$$\mathsf{NextStep}_s(\mathsf{Input}_i, \mathsf{ST}_{i-1}; r) \rightarrow (\mathsf{Output}_i \| P_i, \mathsf{ST}_i)$$

On user input Input_i, previous state ST_{i-1}, and random coins r this function produces a user output Output_i, and new state ST_i. (For some applications described later in this work, we subdivide the output into a *user output* for private delivery to the user, and a *public output* P_i that is intended to be broadcast to a network.)

In our model a (possibly adversarial) User is responsible for invoking each round of computation. A key distinction from previous work (*e.g.,* [NFR+17]) is that the user *adaptively* selects the input to each round of the computation, possibly as a function of previous outputs.[6] We do not require that the User completes a given computation. Instead we seek to achieve the following intuitive properties: (1) for each round, the user learns only the output, but no intermediate function of the computation, and (2) the user cannot *rewind* the functionality or access the state held in between rounds. We seek to achieve these goals even in settings where the Custodian has no secure means to keep state between rounds of computation, and must outsource this function to the User. Finally, as a third goal, we wish to construct a means to condition computation on the *public broadcast* of certain outputs to a consensus network.

Using the contract to keep state. A first idea behind our first construction is to use the Contract as a means to enforce the order of a computation on the (stateless) Custodian; this prevents the adversary from *rewinding* and executing earlier rounds of the functionality on alternative inputs. As an auxiliary benefit, the Contract provides a relatively censorship-resilient channel that the Custodian may access use to receive or transmit data for public consumption. This allows the Custodian to condition operation on network broadcasts as well as other types of public transaction supported by the Contract, such as monetary payments.

A protocol sketch. We begin with a simplified version of our main construction. This approach divides the computation into two portions: one executed by the Custodian and the other by the Contract. Our protocol operates under the key assumption that Contract outputs can be *authenticated*, *e.g.,* by verifying a proof-of-work chain produced by a consensus network. The User acts as a communication channel between the parties, and is responsible for both selecting the input to each round of computation and invoking the Custodian and Contract on specific values. We assume a trusted setup in which the Custodian is provisioned with a computation secret s as well as an additional secret key S sampled uniformly from $\{0,1\}^\lambda$, while the Contract state is initialized to be empty. Let $C_0 \leftarrow \varepsilon, \mathsf{STenc}_0 \leftarrow \varepsilon$.

For $i \geq 1$, we execute the i^{th} round of the computation as follows:

[6] We remark that this is the only appropriate model for a dynamic cryptographic access system, given that the identity of each file to be accessed may depend on the result of previous file accesses.

1. The user first commits to the messages $(\mathsf{Input}_i, C_{i-1})$ using a secure commitment scheme, where Input_i is the user's chosen input to the i^{th} round of computation and C_{i-1} is the commitment generated during the previous round of the protocol (or ε initially).

2. The user transmits the resulting commitment C_i to the smart Contract.[7] The smart Contract records the commitment C_i and accesses its local state to obtain the commitment C_{i-1} that the user transmitted during the previous call to the Contract (or $C_0 = \varepsilon$ at the first round). It then returns the pair (C_{i-1}, C_i) to the User, along with an authenticator π_i that verifies the provenance of the returned pair. In a private consensus network, this authenticator might be a signature calculated over the returned value. In a public consensus network, the authenticator may comprise a proof-of-work computed over the resulting transaction and some number of subsequent blocks.

3. To complete this round of the calculation, the user now invokes the Custodian on the tuple (C_{i-1}, C_i) and π_i, as well as Input_i and the commitment randomness used to formulate C_i. The user also provides the Custodian with an *encrypted state* ciphertext STenc_{i-1} that was produced during the previous call to the Custodian (or ε on the first round).

4. The Custodian verifies the commitment C_i using the given inputs and random coins. It then uses its internal secret S as the key for a pseudorandom function PRF, in order to derive the following three intermediate secrets. These secrets are defined as follows:

   (a) A decryption key for the previous state ciphertext STenc_{i-1} is derived using PRF_S on input C_{i-1}.
   (b) An encryption key for the new state ciphertext STenc_i is derived using PRF_S on input C_i.
   (c) A set of *random coins* r_i for the computation is derived by keying a second pseudorandom generator on S and STenc_i.

   The Custodian decrypts the previous state ciphertext, and if this succeeds it invokes the $\mathsf{NextStep}$ function on the given input, coins and state. Finally, the Custodian encrypts the new state ST_i using the newly generated key. The Custodian outputs Output_i and STenc_i to the user, or the distinguished symbol $\perp$ if any operation fails.

Our main observation is that if the Custodian is honest, then each pair (C_{i-1}, C_i) can be viewed as a commitment to an entire computational trace, including user inputs, random coins and state. Provided that it is infeasible to "forge" an authenticator tag π_i, then the user can *replay* specific inputs to the Custodian (to receive the same output), but can never rewind and execute the Custodian on new inputs. Success in this would require the user to forge the authenticator tag π_i produced by the Contract, or to defeat a cryptographic primitive such as the commitment scheme or authenticated encryption.

Finally, we remark that the protocol above achieves only one of our stated goals: ensuring that the stateless Custodian can perform stateful interactive computation. However, it does not allow the Custodian to publish outputs via the Contract. Adding this capability is simple: we simply create a new field for the Custodian's output P_i, and require that this output be presented to the Contract along with the commitment to the next round's input. The Contract response is simply updated to include the published value. An adversarial user can still censor or even forge the outputs from this Contract; however, from this point she loses the ability to continue the computation.

How to instantiate the Custodian? Thus far we have discussed several mechanisms for instantiating the smart contract. However, the Custodian is more exotic, as it must be capable of securely performing secret computations without revealing any intermediate function of the computation to the user.

In §5 we discuss two main approaches to constructing a viable Custodian functionality. The first, and most practical approach is to use secure hardware. We stress that, as the requirements on the Custodian

[7]In this example we will assume that the User possesses a means to authenticate herself to the Contract. In practice, existing smart Contract systems such as Ethereum provide this natively, by allowing a User to instantiate contracts that embed specific public keys. The User may hold the corresponding signing key.

are relatively simple, the necessary hardware may be surprisingly inexpensive. In particular, this hardware merely needs to embed a single unchanging secret that can be installed at the factory, and requires no ability to retain immutable state between round. Indeed, stateless hardware designs have been widely deployed by manufacturers such as Apple for the purpose of cryptographic file encryption [App16], and these systems have also been shown to be vulnerable to attacks that roll back program state [Sko16].

A more intriguing possibility is that the Custodian might be instantiated using purely *software* techniques. A promising candidate technology for this approach is *witness encryption* (WE) [GGSW13, GKP$^+$13a] as well as a more powerful variant known as *functional witness encryption* (FWE) [BCP13]. There are many caveats to using these techniques, and we offer them only as a conjecture. We explore this area more fully in §5.

1.3 Outline of this paper

The remainder of this paper proceeds as follows. In §2 we define a Custodian-Contract scheme (CCS) and present security definitions for the primitive. In §3 we present our main construction. In §4 we describe several applications of the CCS primitive, including several related to cryptographic access control and MPC fairness, and in §5 we discuss practical instantiations of the CCS components. In §6 we implement our proposals and present experimental results. Finally, §7 surveys related work.

2 Definitions

Notation: Let λ be a security parameter. Let ℓ be a non-negative integer polynomial in λ and let $\nu(\cdot)$ indicate a negligible function. We use the notation $A \overset{c}{\approx} B$ to denote that the distributions A and B are computationally indistinguishable.

Protocol Parties. A Custodian-Contract Interaction is a protocol between three parties: the custodian functionality $\mathcal{C}$, the contract functionality $\mathcal{N}$, and a user $\mathcal{U}$. We now describe the operation of these two elements.

The contract $\mathcal{N}$. The trusted *contract* functionality models a smart contract system. This can be viewed as a stateful functionality that initializes with state $\mathsf{z}_0 = \varepsilon$. For $i \geq 1$, the i^{th} round of computation is invoked on an input selected by the user $\mathcal{U}$ and previous z_{i-1} stored by the contract. If the computation is successful, the contract records a new internal z_i and returns Output_i along with a publicly-verifiable authentication tag π_i. We require that $\mathcal{N}$ has access to a trustworthy broadcast output path, *e.g.,* that once invoked it can broadcast data reliably to the network. We recall that $\mathcal{N}$ does not possess cryptographic secrets, *i.e.,* all of its state is public.

The custodian $\mathcal{C}$. The trusted custodian models a cryptographic obfuscation system or a piece of stateless secure hardware hardwired with a deterministic functionality and is initialized with a pair of secrets s, S. When the custodian is invoked on by the user $\mathcal{U}$, it calculates an output the result (or a distinguished error symbol $\perp$). We require that the description of $\mathcal{C}$'s operations are public , and that the adversary views (and controls) only the inputs and outputs.

The user $\mathcal{U}$. The user is a (possibly adversarial) party that invokes both the custodian and the contract functionalities. The user selects the inputs to each round of computation and receives the *user outputs*. It provides a communication channel between the other parties.

The functionality $\mathcal{F}$. For flexibility we define a slightly more robust computing functionality than the one given in the intuition for this paper. This definition breaks the computation into two distinct algorithms: a *secret* portion $\mathsf{NextStep}$ and a *public* algorithm $\mathsf{DoContract}$ that can be run by the contract.[8] This more complete definition allows us to implement functionalities that require additional capability, such as the

[8]In practice, functionalities may employ a "dummy" contract routine that simply publishes and outputs its inputs.

ability to collect public data (as in [ZCC$^+$16]), or to interface with other smart contracts running on the same contract platform.

More formally, we define the computing functionality $\mathcal{F}$ as a pair $(\mathsf{NextStep}, \mathsf{DoContract})$ where each algorithm is deterministic and has the input/output interface described below:

$\mathsf{NextStep}_s(\mathsf{Input}_i, \mathsf{ST}_{i-1}; r_i) \to (\mathsf{Output}_i \| P_i, \mathsf{ST}_i)$. On input a user input Input_i, a previous state ST_{i-1}, and random coins r_i, this algorithm produces a user output Output_i (including an optional *public output* P_i) and new state ST_i.

$\mathsf{DoContract}(P_{i-1}, U_i, \mathsf{z}_{i-1}) \to (D_i, \mathsf{z}_i)$. On input a previous custodian's public output z_{i-1} and previous contract state z_{i-1}, outputs a result D_i and updated contract state z_i.

2.1 Custodian-Contract Scheme

A Custodian-Contract scheme Σ_{ccs} consists of a tuple of algorithms $(\mathsf{Setup}_{\mathcal{F}}, \mathsf{ExecuteCustodian}_{S,\mathcal{F}}, \mathsf{ExecuteContract}_{\mathcal{F}}, \mathsf{ExecuteUser}_{\mathcal{F}})$. The interface to these algorithms is given in Figure 2.

$\mathsf{Setup}_{\mathcal{F}}(1^\lambda)$. On input a security parameter λ and a functionality $\mathcal{F}$, outputs the secrets (S, s) for the custodian and initializes $\mathsf{STenc}_0 \leftarrow C_0 \leftarrow \varepsilon$.

$\mathsf{ExecuteCustodian}_{S,s,\mathcal{F}}(\mathsf{Input}_i, \mathsf{STenc}_{i-1}, r_i, O_i, \pi_i)$. This deterministic algorithm is parameterized by the secrets (S, s) and a functionality $\mathcal{F}$. At the i^{th} round, this algorithm takes a program input Input_i and an encrypted program state STenc_{i-1} along with the commitment randomness r_i and an output from the contract (O_i, π_i). It outputs a new encrypted state along with user and public outputs $(\mathsf{STenc}_i, \mathsf{Output}_i, P_i)$

$\mathsf{ExecuteContract}_{\mathcal{F}}(C_i, U_i, P_{i-1}, \mathsf{z}_{i-1})$. This algorithm is parameterized by a functionality $\mathcal{F}$. At the i^{th} round, this algorithm takes a commitment C_i, an optional user input to the contract U_i, and a previous custodian output P_{i-1} from the user along with previous contract state z_{i-1}. It outputs a tuple (O_i, π_i).

$\mathsf{ExecuteUser}_{\mathcal{F}}(\mathsf{Input}_i, C_{i-1}, r_i)$. This algorithm is parameterized by a functionality $\mathcal{F}$. At the i^{th} round, this algorithm takes a value Input_i produced by the user, and outputs a string C_i.

Figure 2: Definition of a Custodian-Contract scheme (CCS).

The Custodian-Contract protocol. A Custodian-Contract scheme is used to instantiate an interactive protocol between the Custodian, User and Contract. Figure 4 illustrates a single round of this protocol. To initialize each interactive computation, a trusted party first runs Setup on input the functionality $\mathcal{F}$ and a security parameter, to generate the secrets (S, s) used by the custodian. The contract and custodian previous states are initialized to ε. For each round $i \geq 1$ the protocol now proceeds as follows:

1. The user $\mathcal{U}$ first runs the algorithm $\mathsf{ExecuteUser}$ on a first input Input_i, to produce a commitment C_i.
2. The user then invokes the contract functionality $\mathsf{ExecuteContract}$ on C_i, their own input U_i, and a previous custodian output P_{i-1} (if available), to produce the tuple (O_i, π_i).
3. To complete the round of computation, the user now invokes $\mathsf{ExecuteCustodian}$ on the program input, previous encrypted state (if available) and the output of the contract functionality.

2.2 Correctness and Security

Correctness. Intuitively, correctness for a Custodian-Contract scheme is defined in terms of the parameterized functionality $\mathcal{F}$. At each step of the Custodian-Contract interaction, the contract and custodian should output the same outputs as an oracle implementing $\mathcal{F}$.

Security. We define security for a Custodian-Contract interaction using a real/ideal-world definition. This definition specifies two experiments: a **Real** experiment in which the adversarial user interfaces with oracles that implement the custodian and contract functionalities respectively, and an **Ideal** experiment that models the correct and stateful operation of the underlying functionality $\mathcal{F}$. Because we plan to implement the contract functionality on the blockchain, the authenticity of contract outputs are protected economically. In this setting it is possible that a powerful attacker will succeed in forging a small number of contract calls, each one at a great expense. Because we cannot prevent these forgeries altogether, we require that the security of a scheme minimally degrades "gracefully" in the face of such forgeries. We address this using following "best possible" definition. An attacker who forges q_{forge} separate contract outputs obtains the same utility as an attacker who is able to rewind and execute the $\mathsf{NextStep}$ algorithm exactly q_{forge} times.

Intuitively, our goal is to ensure that even an adversarial user $\mathcal{U}$, with oracle access to a forgery oracle F, operating the custodian and contract cannot learn any information beyond their normal, expected inputs and outputs. The forgery oracle will output a validating authenticator for π on up to q_{forge} inputs from the user. We define security using a simulation-based definition: we require that for all p.p.t. adversaries $\mathcal{U}$ in the **Real** experiment, there exists a simulator $\mathcal{S}^{\mathcal{O}^{\mathcal{F},S,F}(\cdot)}$ that plays the **Ideal** experiment, such that no p.p.t. algorithm can distinguish the output of $\mathcal{U}$ from the output of $\mathcal{S}$ (See Appendix A for formal definitions of both experiments).

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

Definition 2.1 (Simulation security) *We say that a Custodian-Contract scheme* Σ_{ccs} *is* simulation secure *for functionality* $\mathcal{F}$ *if* $\forall$ *p.p.t. real-world adversarial users* $\mathcal{U}$ *and* $\forall$ *non-negative integers* q, q_{forge}*, there exists a p.p.t. ideal-world user* $\mathcal{S}^{\mathcal{O}^{\mathcal{F},S,F}(\cdot)}$ *such that* $\mathbf{Real}(\Sigma_{\mathsf{ccs}}, \mathcal{F}, \mathcal{U}, q, q_{\mathsf{forge}})$ *is computationally indistinguishable from* $\mathbf{Ideal}(\mathcal{F}, \mathcal{S}^{\mathcal{O}^{\mathcal{F},S,F}(\cdot)}, q, q_{\mathsf{forge}})$.

Privacy against third parties. The definitions above address the security of a Custodian-Contract Interaction scheme against a dishonest user. However, we must also provide privacy for an *honest* user against third parties who may have read access to the blockchain. We address this using a separate *user privacy* definition. Intuitively, under the assumption of an honest contract and user, we require that a third party should learn nothing from the inputs and outputs of the contract beyond what the user would normally learn from the public outputs of the computation. We address this more fully in Appendix A.1.

3 Our Construction

```
Algorithm 1: Setup
  Data: Input: $1^\lambda$
  Result: Output: Secret $S$
  $s \xleftarrow{\$} \{0,1\}^m, S \xleftarrow{\$} \{0,1\}^\ell, C_0 \leftarrow \varepsilon, \mathsf{STenc}_0 \leftarrow \varepsilon, \mathsf{z}_{i-1} \leftarrow \varepsilon$
  $\mathsf{pp} \leftarrow \mathsf{CSetup}(1^\lambda)$
  Output $\mathsf{pp}, s, S, C_0, \mathsf{STenc}_0, \mathsf{z}_{i-1}$
```

```
Algorithm 2: ExecuteUser
  Data: Input: $(\mathsf{Input}_i, C_{i-1}, r_i)$
  Result: Output: $C_i$
  // Commit to the user input and previous commitment
  $C_i \leftarrow \mathsf{Commit}(\mathsf{pp}, \mathsf{Input}_i \| C_{i-1}; r_i)$
  Output $C_i$
```

```
Algorithm 3: ExecuteContract
  Data: Input: $(C_i, P_{i-1}, U_i)$, State: $(C_{i-1}, \mathsf{z}_{i-1})$
  Result: Output $(O_i, \pi_i)$ or $\perp$
  $(D_i, \mathsf{z}_i) \leftarrow \mathsf{DoContract}(P_{i-1}, U_i, \mathsf{z}_{i-1})$
  $O_i \leftarrow (C_{i-1} \| C_i \| D_i \| P_{i-1})$
  Compute $\pi_i$ over $O_i$
  Output $(\pi_i, O_i)$
```

```
Algorithm 4: ExecuteCustodian
  Data: Input: $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, [\pi_i, O_i])$,
        Secrets: $S, s$, Max state length: $n$
  Result: Output: $(\mathsf{STenc}_i, \mathsf{Output}_i)$ or $\perp$
  $\mathsf{Assert}(\mathsf{Verify}(O_i, \pi_i) = 1)$
  Parse $O_i \rightarrow (C'_{i-1} \| C'_i \| D_i \| P_{i-1})$
  $\mathsf{Assert}(C'_i = \mathsf{Commit}(\mathsf{pp}, \mathsf{Input}_i \| C_{i-1}; r_i))$
  if $\mathsf{STenc}_{i-1} \neq \varepsilon$ then
      $k_{i-1} \leftarrow \mathsf{PRF}_S(\text{“enc”} \| C'_{i-1})$
      $\mathsf{ST}_{i-1} \leftarrow \mathsf{Unpad}(\mathsf{Decrypt}(k_{i-1}, \mathsf{STenc}_{i-1}))$
  else
      if $C_{i-1} = \varepsilon$ then
          $\mathsf{ST}_{i-1} \leftarrow \varepsilon$
      else
          Halt and Output $\perp$
  // Perform the computation
  $r_i \leftarrow \mathsf{PRG}(\mathsf{PRF}_S(\text{“rand”} \| C'_i))$ ;
  $(\mathsf{Output}_i, \mathsf{ST}_i, P_i) \leftarrow \mathsf{NextStep}_s(\mathsf{Input}_i \| D_i \| P_{i-1}, \mathsf{STenc}_{i-1}; r_i)$ ;
  $k_i \leftarrow \mathsf{PRF}_S(\text{“enc”} \| C'_i)$
  $\mathsf{STenc}_i \leftarrow \mathsf{Encrypt}(k_i, \mathsf{Pad}_n(\mathsf{ST}_i))$
  Output $(\mathsf{STenc}_i, \mathsf{Output}_i, P_i)$
```

Figure 3: Our main construction Σ_{ccs}.

In this section we present a specific construction of a Custodian-Contract scheme. Our construction makes use of the following cryptographic primitives. In addition to the standard cryptographic primitives, we define an abstract “authenticator” primitive that will be used to authenticate the validity of a consensus transcript. We discuss this final primitive in more detail below.

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.

Authenticated symmetric encryption. We require a symmetric encryption scheme consisting of the (possibly probabilistic) algorithms $\Sigma_{\mathsf{AE}} = (\mathsf{Encrypt}, \mathsf{Decrypt})$ where each accepts a key uniformly sampled from $\{0,1\}^\ell$. We require a scheme that meets the AE definition of Rogaway [Rog02]. This requires that a ciphertext is indistinguishable from a random bitstring even in a model where the attacker can query encryption and decryption oracles. We also employ a padding scheme $(\mathsf{Pad}_n, \mathsf{Unpad})$ that pads a plaintext to some chosen bitlength n.

Pseudorandom Functions and Generators. Our construction uses a pseudorandom function family (PRF) PRF that in input a seed and some value in $\{0,1\}^*$ outputs a string in the range $\{0,1\}^\ell$ as well as a pseudorandom generator (PRG) that on input a seed in $\{0,1\}^\ell$ outputs a stream of pseudorandom bits.

Contract authenticators. Finally, we employ a primitive that we refer to as a *contract authenticator*. An authenticator is a publicly verifiable authentication primitive that allows any party to verify the authenticity of a message issued by the contract.[9] In public consensus networks, the authenticator may be constructed using a computational proof-of-work (see §5) computed over each contract output. We require that it is difficult to forge a pair a new pair (m, π) even when the attacker is given access to an oracle for authenticating chosen messages. We refer to this definition as SUF-AUTH (this is analogous to the SUF-CMA definition used for signatures). We also consider a variant of this definition called q_{forge}-SUF-AUTH in which the adversary is allowed to produce at most q_{forge} forged messages.

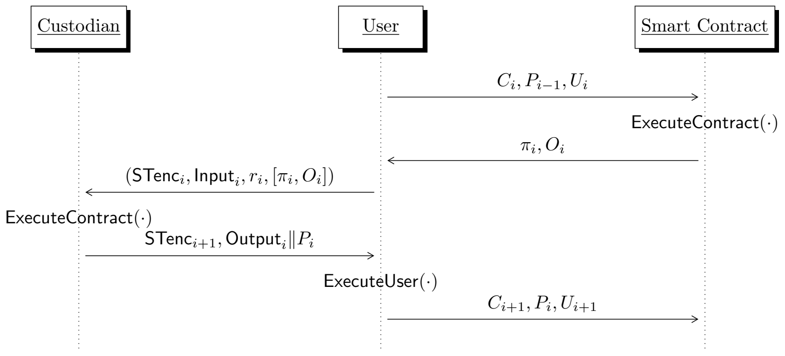

Figure 4: Illustration of a single round of a Custodian-Contract interaction. The User runs the ExecuteUser algorithm on a computation input Input_i and random coins r_i, then invokes the Contract on the resulting commitment C_{i-1}, their own input U_i, and an (optional) previous Custodian *public output* P_{i-1}. The contract returns an authenticator/output pair (π_i, O_i). The User now invokes the Custodian on all of the above values as well as a previous *encrypted state* STenc_{i-1} to produce a user output Output_i, a new public output P_i, and a new encrypted state STenc_i.

3.1 Main Construction

We now present our main construction for a Custodian-Contract scheme (CCS) and address its security. Recall that a CCS consists of four algorithms with the interface described in 2. We present each of these algorithms in Figure 3.

We address the correctness of the scheme in Appendix B. We now proceed to our main security theorem.

Theorem 3.1 If Σ_{com} is a secure commitment scheme; Σ_{AE} satisfies the strong AE definition of [Rog02]; PRF and PRG are each pseudorandom; and contract authenticators are q_{forge}-forgeable, then for all allowed functionalities $\mathcal{F}$ and $q_{\mathsf{forge}} \geq 0$ the scheme $\Sigma = (\mathsf{Setup}, \mathsf{ExecuteCustodian}, \mathsf{ExecuteContract}, \mathsf{ExecuteUser})$ presented in Figure 3 satisfies Definition A.1.

We present a proof of Theorem 3.1 in Appendix B.

4 Applications

We now describe several applications that use Custodian-Contract Interaction, and present the relevant implementations for each. Each proposed application employs the main construction we presented in §3 (depicted in Figure 3) to implement a specific functionality. We present algorithms of the functionality $\mathcal{F} = (\mathsf{NextStep}, \mathsf{DoContract})$ required to implement the specific applications.

[9]Indeed, in *private* consensus networks the authenticator may be constructed using an SUF-CMA digital signature scheme with some pre-generated keypair (pk, sk) specific to the contract.

4.1 File Access Logging

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

In place of a trusted server, we propose to use CCS to implement mandatory logging for protected files. In this application, a local Custodian stores a master key for some collection of files, *e.g.,* a set of files stored on a device.[10] The Custodian then employs the *public output* property of the consensus network to ensure that prior to each file access the contract must write a public log identifying the file access to the network. To provide confidentiality of file accesses, the Custodian may encrypt the log entry under the public key of some auditing party and will only release the decryption once this ciphertext is published.[11] We present the NextStep and DoContract algorithms for a simple two-phase version of this proposal in Algorithms 5 and 6.

4.2 Strong Encrypted Backups from Weak Passwords

In the past several years, device manufacturers have widely deployed end-to-end file encryption for devices such as mobile phones [App16, Pro] and cloud backup data [Las, Krs16]. Increasingly, these systems require users to hold their own secrets rather than trusting them to the manufacturer.

Implementing end-to-end encryption systems at scale poses a dilemma for system designers. Encryption systems require high-entropy cryptographic keys, but users are prone to lose or forget high-entropy passwords. To address this tension, manufacturers have increasingly turned to the use of *trusted hardware* such as on-device cryptographic co-processors [App16, ARM] and cloud-based HSMs [Krs16] for backup data. In this model a user may authenticate using a relatively weak device passcode such as a PIN, and the hardware will then release a strong encryption key. To prevent passcode guessing attacks, this *stateful* hardware throttles or limit the number of login attempts.[12]

A Custodian-Contract Scheme provides an alternative mechanism for limiting (or throttling) the number of guessing attempts on password-based encryption systems. A user can employ an inexpensive USB drive or an extremely low-cost embedded chip with no requirement to keep state (*e.g.,* an FPGA) to host a simple Custodian, with an internal secret key K. The custodian is constructed to release K only when it sees the proper output of a smart contract on the ledger *and* receives a valid input. The smart contract in turn may limit the number of guesses an adversary can make within a given period. This Custodian and Contract functionalities required to enable this functionality are relatively simple, under the assumption that the Contract has access to a clock equivalent.[13] For reasons of space we do not formally define the functionality $\mathcal{F}$ required to implement this application; it is very similar to the functionality of §4.1.

We remark that in practice the decryption process in such a system can be fairly time consuming, due to the need to interact with the consensus network and obtain a number of blocks of valid output prior to running the Custodian. This system may be useful for low frequency applications such as recovering encrypted backups or emergency password recovery.

4.3 Autonomous Ransomware

Ransomware is increasingly becoming a problem in hospitals and other corporate environments [Zet16]. These systems consist of a malware payload that encrypts user files, and an online C&C system that delivers

[10] If the Custodian is implemented using cryptographic techniques such as FWE, a unique Custodian 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 Custodian when the files are created or provisioned onto the user's device.

[11] The corresponding decryption key may be held offline.

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

[13] In the case of a public consensus network such as Ethereum or Bitcoin, this clock can be implemented by requiring at least K empty blocks between each transaction.

Algorithm 5: File Access Logging ($\mathsf{NextStep}$)

```
Data: Input: $\mathsf{Input}_i$ (filename), $\mathsf{ST}_{i-1}, D_i, P_{i-1}; r_i$; Secret: PRF Key $K$;  Constants: $pk_{\mathsf{auditor}}$
Result: $\mathsf{Output}_i$ or $\perp$
Parse $\mathsf{ST}_{i-1}$ as $(\mathsf{phase}, CT, \mathsf{filename})$
if $\mathsf{phase} = 2$ then
    Phase 2: Verify contract response
    if $D_i = (\text{“}\mathsf{Allow}\text{"} \parallel CT)$ then
        // Output a decryption key for filename
        $\mathsf{ST}_i = (1, \cdot, \cdot)$
        output $(\mathsf{Output}_i \| P_i) = (\mathsf{PRF}_K(\mathsf{filename}) \| \varepsilon)$
    else
        // Failure
        output $\perp$
else
    // Phase 1: Encrypt the filename to be logged
    $CT \leftarrow \mathsf{PKEnc}(pk_{\mathsf{auditor}}, \mathsf{Input}_i)$
    $\mathsf{ST}_i \leftarrow (2, CT, \mathsf{Input}_i)$
    output $(\mathsf{Output}_i \| P_i) = (\varepsilon \| CT)$
```

Algorithm 6: File Access Logging ($\mathsf{DoContract}$)

```
Data: Input: $(C_i, P_i)$, Constants: $R$
Result: $D_i$
write $P_i$ as public contract output
output $D_i = \text{"}\mathsf{Allow}\text{"} \| P_i$
```

decryption keys to customers who pay a ransom – typically in a cryptocurrency such as Bitcoin. In current ransomware, the system that deliver keys represents a weak point in the ransomware ecosystem. This delivery exposes ransomware operators to tracing [Tec16]. As a result, some operators have fled without key material, or have deployed non-functional "dummy" ransomware.

We now consider a potential *destructive* application of the CCS paradigm: the development of *autonomous* ransomware that guarantees decryption without the need for any 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 within a Custodian. To activate the ransomware, the user runs the Contract portion of the malware on a payment network such as Bitcoin or Ethereum, passing as input the identity of a valid payment transaction to the ransomware operator's address. The Custodian then validates the result and authenticator returned by the Contract – which in this case comprises a chain of computationally expensive proofs of work – and delivers the key K if this proof is valid. Algorithms 7 and 8 present a simple example of the functionality.

We note that the Custodian 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 (see §6 for more discussion). 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.[14]

Algorithm 7: Ransomware (NextStep)

```
Data: Input: $\mathsf{Input}_i, \mathsf{ST}_{i-1}, D_i, P_{i-1}; r_i$, Secret: Key $k$
Result: $\mathsf{Output}_i$ or $\perp$
if $D_i =$ "Decrypt" then
    output $\mathsf{Output}_i = k$;
else
    output $\perp$;
```

Algorithm 8: Ransomware (DoContract)

```
Data: Input: $(C_i, U_i)$, Constants: address, amount
Result: $D_i$
parse $U_i$ to obtain TransID
// Use the following call to verify the payment
if ValidatePayment(TransID, amount, address) $= 1$ then
    output $D_i =$ "Decrypt";
else
    output $D_i =$ "Invalid";
```

4.4 "Fair Encryption" and MPC

Multi-party Computation (MPC) [Yao86, GMW87] guarantees that parties $1, \ldots, n$ can run a function f on private inputs $x_1, \ldots x_n$ without revealing the inputs (or any intermediate function) to the other parties. A known limitation of general MPC is the difficulty of obtaining *fairness* when at least half of the parties are malicious [Cle86]: intuitively, the malicious parties may abort the protocol after learning the result, but

[14]See Lewi *et al.* for a survey on the current practical limitations of this technology [LMA+16].

before the honest parties have learned it. This can be catastrophic in applications such as financial trading. Efforts to address this problem have led to relaxations in security properties [GK10, Pin03, BLOO11].

Recently Gordon [GK10] *et al.* and Andrychowicz *et al.* [ADMM14] proposed a partial solution that uses consensus networks to *penalize* users who abort. This approach assumes that the parties can correctly select a penalty that will compensate the honest users for the protocol failure. Unfortunately, choosing such a penalty may not be feasible in practice: for example, if the utility of malicious behavior is difficult to calculate, or if the capital required to participate in the protocol exceeds any party's resources. In these cases it would be preferable if the consensus network could actually force the protocol to complete.

Fair encryption. As a building block for solving this problem, we propose *fair encryption.* Intuitively, this primitive ensures that if one authorized decryptor can decrypt a ciphertext, then *all* authorized decryptors may do so as well. We accomplish this by constructing a custodian that will decrypt a CCA2-secure ciphertext if and only if a valid ciphertext has been placed on the public consensus network transcript by a Contract.

Algorithm 9: Fair Encryption (NextStep)

```
Data: Input: $\mathsf{Input}_i, \mathsf{ST}_{i-1}, D_i, P_{i-1}; r_i$, Secret: Key $k$. Constants: Ciphertext hash $H$
Result: $\mathsf{Output}_i$ or $\perp$
if $\mathsf{Input}_i$ = "GenKey" then
    $(pk, sk) \leftarrow \mathsf{PKMGen}(1^\lambda)$
    $\mathsf{ST}_i \leftarrow sk$
    output $\mathsf{Output}_i = pk$
else if $\mathsf{Input}_i$ = "Decrypt"$\|CT$ AND $\mathsf{ST}_{i-1} \neq \varepsilon$ then
    Parse $\mathsf{ST}_{i-1}$ as $sk$
    $M \leftarrow \mathsf{PKMDec}(sk, CT)$
    output $M$
```

Algorithm 10: Fair Encryption (DoContract)

```
Data: Input: $(C_i, P_i, U_i)$. Contants: $pk_{mpc}, pk_0, pk_1 \ldots pk_{n-1}$. Init: Set $S \leftarrow \phi$, Hash $H \leftarrow \varepsilon$
Result: $D_i$
write $P_i$ as public contract output
output $D_i$ = "Decrypt"$\|P_i$
```

As a building block, fair encryption requires a multi-key public key encryption system (PKMGen, PKMEnc, PKMDec) with the following properties. Given n public keys $pk_1, \ldots, pk_n$, the encryption routine produces a ciphertext that can be decrypted using any *one* of the corresponding secret keys. The resulting ciphertext must be non-malleable, *i.e.,* the (honest) decryption routine must output $\perp$ on input any valid secret key and a mauled version of the ciphertext. Such a scheme can be easily built using common techniques; for completeness we include one such construction in Appendix C.

To use a fair encryption scheme, each party configures an honest instance of the custodian. At setup time, each Custodian outputs a public key pk for the encryption scheme above. To produce a ciphertext for all parties, the sender encrypts a single message to all public keys in the set $pk_1, \ldots, pk_n$.[15] On receiving the resulting ciphertext CT, a receiving User must invoke the Contract, which in turn places CT onto the public transcript produced by the consensus network (subsequent users may simply provide a reference to the first publication). When the User obtains the authenticated result, she presents it to her Custodian which verifies

[15]This scheme requires a non-malleable ciphertext that encrypts to any set of public keys $pk_1, \ldots, pk_n$ and allows decryption with any one of the corresponding secret keys. The scheme must also be non-malleable, so that an attacker cannot *maul* the ciphertext so that only a subset of the honest decryptors will be able to decrypt it correctly.

the integrity of the ciphertext and Contract output before decrypting using sk.[16] We present an instantiation of the functionality in Algorithms 9 and 10.

MPC from Fair Encryption. To achieve fairness in an MPC protocol, each party configures an honest Custodian and generates a public key pk. Rather than computing a function $f(x_1, \ldots, x_n)$ directly, the parties collaborate to compute $\mathsf{FairEncrypt}_{pk_1,\ldots,pk_n}(f(x_1, \ldots, x_n))$. At the conclusion of this protocol, each party submits the resulting ciphertext CT to the Contract in order to obtain its decryption. Even if the resulting MPC computation is unfair, a malicious party will be forced to transmit their result to the Contract prior to learning the output of the computation. Because this transmission produces a public record on the consensus transcript, all parties will necessarily learn the result.

We observe that if the Custodian is implemented using a form of (Extractable) Witness Encryption (WE) as in [Jag15], the protocol is simpler in that there need be only one Custodian and one "public key": the relation encoding the consensus authenticator verification statement. Even when the Custodian is implemented in trusted hardware (such as Intel SGX enclaves) the resulting construction still does not significantly relax the underlying security of the MPC protocol. In this case, the Custodian is trusted only with the *fairness* of the protocol: a fully corrupted custodian cannot violate either correctness or input privacy.

5 Realizing the Custodian and Contract

In this section we turn to the matter of instantiating the custodian and contract components of a real system.

5.1 Realizing the Custodian

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

While each of the environments above provides immutable statekeeping, a key feature in our proposal is that our approach does not require a secure means for recording state within the Custodian. (Indeed, historically some access control systems have failed because they admitted state rollback attacks [Sko16].) This may enable the construction of stateful functionalities using relatively inexpensive secure hardware, such as a processor with an embedded decryption key, or even using FPGAs as a building block. Along these lines, Nayak *et al.* [NFR+17] recently showed how to build trusted non-interactive Turing Machines from minimal *stateless* trusted hardware. Such techniques open the way for the construction of arbitrary Custodian functionalities on relatively inexpensive hardware.

Witness encryption. Witness encryption (WE), first proposed by Garg, Gentry, Sahai, and Waters [GGSW13], allows for the encryption of of data such that it can only be opened if the decryptor knows a witness to the statement. Witness encryption defined for an NP language $\mathcal{L}$ and witness relation R. It allows a user to encrypt a message m under a instance x. Decryption of the resulting ciphertext is possible only if $x \in \mathcal{L}$ and. It can be built from Indistinguishability Obfuscation [GGH+13].

Witness Encryption seems an obvious candidate for constructing the Custodian, at least for some functionalities. The limitation of WE is that this construction only guarantees the ciphertext is secure in the case where $x \notin L$ *i.e.* there is no witness. Of course, when considering proof-of-work consensus systems there is always a valid proof of work, it is merely computationally difficult to build it. To address this we require a stronger notion of *extractable* witness encryption (EWE) which requires that anyone decrypting the ciphertext actually know the witness, not merely that such a witness exist. Goldwasser *et al.* conjecture that the original WE construction of Garg *et al.* may possess this property under stronger cryptographic assumptions [GKP+13b].

[16] This scheme requires a non-malleable ciphertext that encrypts to any set of public keys $pk_1, \ldots, pk_n$ and allows decryption with any one of the corresponding secret keys. The scheme must also be non-malleable, so that an attacker cannot *maul* the ciphertext so that only a subset of the honest decryptors will be able to decrypt it correctly.

Functional witness encryption (FWE). FWE [BCP13, ABG+13] is a more powerful conjectured primitive in which the decryption of a ciphertext is not simply a message m, but $f(m, \omega)$ where ω is a witness and f is a function chosen by the encryptor. Using this primitive, we can construct powerful Custodians. However, FWE requires exists under stronger cryptographic assumptions, and the literature contains implausibility results [GGHW13] similar to the impossibility results known for Virtual Black Box obfuscation [BGI+01]. But in both cases, the impossibility results stem from malicious auxiliary input and "contrived" functions. It remains an open question if these results disqualify the existance of FWE for realistic and practical functions; Virtual black box obfuscation, from which FWE can be built, is still an ongoing active research area [BGK+13, BR13].

5.2 Realizing the Contract

A number of different systems may be used to instantiate the Contract. In principle, any stateful functionality capable of producing SUF-CMA signatures can be used for this purpose, including trusted servers and "private blockchains" consisting of a small quorum of servers. However in this work we focus primarily on *public consensus networks*, which are primarily implemented using blockchains.[17]

Security of the contract. An obvious solution for instantiating the Contract is to use a public cryptocurrency (or smart contract) such as Bitcoin or Ethereum. This avoids the need for any globally trusted third party. The limitation of this approach is that these systems cannot offer cryptographic authentication of Contract outputs. Instead, the guarantee provided is *economic* in nature: since these networks authenticate legitimate blocks by attaching computationally-expensive *proofs of work*, the cost of forgery may be high. The exact details of this cost depend on the application and, crucially, the number of simultaneous instances of the system sharing a single consensus network. In the following section we briefly address this analysis.

Forging blockchain fragments. Blockchains such as Ethereum and Bitcoin employ hash-based proofs of work to authenticate blocks. Since these proofs of work are also used for currency "mining", a simple way to judge the cost of forging a block is to evaluate the financial opportunity cost of spending hashpower on such forgeries rather than on mining.

As of early 2017, forging a single Bitcoin block represents an opportunity cost of at least \$12,500 USD in forgone block reward [coi]. Since a Contract may be authenticated by a chain of such blocks (where each block implicitly hashes the previous block), forging a chain of approximately 24 hours worth of block confirmations represents a forgone mining cost of approximately \$1.8M USD.

The challenge in this simple analysis is that a single blockchain fragment may be used by multiple instances of a given Contract. 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 contracts, this fixed cost could be negligible. For scenarios where we expect sufficient instances for this attack to be practical, it is necessary to rate limit the number of contracts included in a given block that the custodian will accept results from.For example, to lower-bound the cost of such an amortization attack to \$1000 USD per instance, it is necessary to limit the number of transactions to 1800 such transactions per day. This may be sufficient protection for applications such as ransomware, where the ransom itself can be priced significantly lower. In the full version of this work we include a more complete economic analysis.

6 Prototype Implementation

To validate our approach we implemented a Custodian-Contract scheme using Intel SGX as a custodian and the Ethereum smart contract system as our ledger. Due to its ubiquity in newer Intel processors, SGX represents a secure computation technology that is both practical and available today. Realizing the Custodian effectively in SGX provides a means for this paradigm to be quickly adopted.

[17] For more background on this technology, see *e.g.,* [TS15, BMC+15].

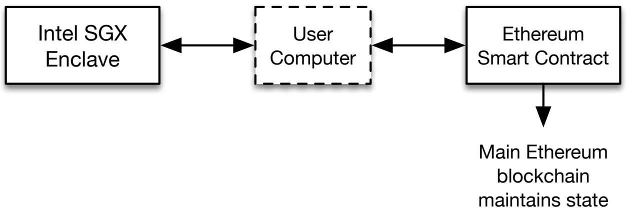

Figure 5: Our prototype implementation of the Custodian-Contract Interaction

To implement the Custodian, we adapted the verification and parsing subset of the C++ Ethereum client [cpp] to work within an SGX enclave running on Linux. Due to limitations placed on code that runs within an SGX enclave, this adaptation was non-trivial. It required the removal of all input/output and downgrading the C++11 client to use the adapted C++03 standard library supplied by the SGX software development kit. This downgrade process was replicated for the subset of Boost on which the Ethereum client relies. The Ethereum client also uses *cryptopp* to perform SHA3 computations; we chose to replace *cryptopp* with a simpler but less efficient standalone implementation of SHA3. Because SGX does not natively support public key cryptography, we also integrated the standalone TweetNacl library [Twe]. Finally, SGX does not support copying C++ objects between userland and a secure enclave, so all input data had to be serialized into byte streams. We built a thin wrapper around the serialization format supported by Ethereum, called Recursive Length Prefix.

We implemented three SGX enclaves: (1) a generic blockchain verifier, (2) a sample ransomware enclave, and (3) a guaranteed logging enclave. We made a number of application-specific optimizations for each enclave that allowed us to omit parts of the protocol in §3. Each enclave was hardcoded a genisis block as ground truth. Finally, we included the capability for the enclave to verify the blockchain fragments passed in by a user against a "genesis block" used as a trusted ground truth. We also provide a trusted *construct* call to allow the party configuring the enclave to set a more recent block as ground truth. In the ransomware application, having the ability to set ground truth at the time of infections allows for faster blockchain verification and as protects against amortization attacks described in §5.2.

Ethereum supports a fast-sync protocol which allows new nodes to quickly download the blockchain by choosing to check the proof of work only on random blocks. We borrowed this protocol for our enclaves and enforced that when a single block fails the proof of work verification, the enclave will reject the fragment. Because the cost of forging a transaction is proportional to the number of blocks that follow it and their difficulties, it is also not particularly important to verify each block preceding the block with the contract ouput. Thus, our enclave simply verifies the proof of work on every block following that block and ensures that the block difficulty is sufficiently high. [18]

Measurements and Test Cases While the Bitcoin proof of work can be verified using only two SHA2565 operations per block, verifying Ethereum's proof of work is an extremely computationally expensive operation that necessitates many SHA3 calls. Modern Ethereum clients alleviate this problem by constructing large RAM caches to amortize this computation cost over many blocks in the chain. Because our implementation is a prototype that is constrained by SGX's 128MB memory limit, we did not to optimize our implementation in this way. This leaves our prototype slower than normal Ethereum clients. Figure 6 shows the running times of various parts of our implementation. In addition to the timing results we gathered from running our enclaves, we also implemented and benchmarked SHA256 in SGX to estimate the block verification time a custodian would take if we had instead used Bitcoin.

[18]The difficulty is a variable field that determines the complexity of the proof of work required for each block. Since this field can change we require our blockchain verifier to adjust the number of blocks required for a proof of work, to avoid forged fragments that claim an artificially low difficulty.

Computation Section	Running Time	Percentage
Verification (Prototype SHA)	**18.35s**	**100%**
SHA3-512	18s	98.8%
Hash Memory Managment	14.62ms	0.0%
Other Computation	.22s	1.1%
Verification (Cryptopp)	**2.27s**	**100%**
SHA3-512	2.14s	94.2%
Hash Memory Managment	7.5ms	0.0%
Other Computation	.13s	5.8%
SGX Overhead	**1.87s**	**100%**
Enclave Setup	1.80s	96.3%
Deserialization	.07s	3.7%
Verification (Bitcoin)	**1.6μs**	**100%**

Figure 6: Table showing the computation times of custodian code. Verification times shown are for a single block

We emphasize that our implementation is a prototype; the difficulties of engineering under the limitations of the current SGX system led to design choices which are not optimal.

7 Related Work

Intel's SGX. Intel's Software Guard Extensions (SGX) allow for the creation of *enclaves*, a trusted execution environment [sgx, MAB+13]. These enclaves can be statically disassembled, but are opaque while running. Enclaves can also *attest* to their current state, proving correct execution [JSR+16, AGJS13]. SGX can cryptographically *seal* data to be used across multiple invocations – corresponding to the encrypted state used in our protocol. Data can either be sealed to the owner of the enclave or just to the specific enclave instance [AGJS13, B16]. Production enclaves can only be produced by obtaining a production license and signing key from Intel, which whitelists licensee's key [Rao16, Int16].

Ethereum and smart contracts. Ethereum is a modern blockchain technology, built on a P2P network, that supports both traditional payment applications as well as distributed consensus computing. Transactions can be direct at users or at smart contracts, in which case miners will run the functionality within the contract. For the output to be saved, each miner runs the function until its termination, consuming *gas* for each instruction executed. If all miners reach a consensus on the function output, the output and state is saved. In any other situation, including insufficient gas, the miners terminate and return the contract to its previous state. Ethereum and other smart contract systems contain many more features which we do not describe, but can be found in [Ethb].

Other applications of smart contracts. Smart contracts are small programs executed autonomously by the blockchain. Examples include Town Crier [ZCC+16] which leverages Intel's SGX to provide an authenticated data feed which can bridge the gap between already existing websites and smart contracts. In [TZL+17], Tramer et al. create the notion of a sealed-glass proof, which leverages Intel's SGX to provide confidentiality and integrity assurances even in the face of side-channel leakage. In a more nefarious setting, Juels et al. introduce the concept of a criminal smart contract which could help enable the (anonymous) commission of a wider range of criminal acts [JKS16].

Ransomware. Ransomware is a class of malware that encrypts data on a victim's system, offering decryption in exchange for payment of a monetary ransom. First proposed by Young *et al.* in 1996 [YY96], as of 2016 ransomware infections have produced an estimated $1 billion in direct and indirect costs [FG16]. The idea of malware and trusted hardware is not a new one. In 2011 Dunn et al. [DHWW11] proposed the idea of hiding malware using a trusted platform module (TPM).

Distributed logging. Secure, distributed logging is an important area of systems research that has been active for a long time [SK98, PPW13, VWGP12, GMMP12]. Of particular note to our work is the commerical product Guardtime [Gua] based on Keyless Signatures' Infrastructure [BKL13]. This project uses ledgers to record events and is able to similarly record log statements.

References

[ABG+13] Prabhanjan Ananth, Dan Boneh, Sanjam Garg, Amit Sahai, and Mark Zhandry. Differing-inputs obfuscation and applications. Cryptology ePrint Archive, Report 2013/689, 2013. `http://eprint.iacr.org/2013/689`.

[ADMM14] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 443–458. IEEE, 2014.

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

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

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

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

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

[BCD+14] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovation with pegged sidechains. Available at `https://blockstream.com/sidechains.pdf`, October 2014.

[BCP13] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability (a.k.a. differing-inputs) obfuscation. Cryptology ePrint Archive, Report 2013/650, 2013. `http://eprint.iacr.org/2013/650`.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. On extractability obfuscation. In *TCC '11*, pages 52–73, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

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

[BGK+13] Boaz Barak, Sanjam Garg, Yael Tauman Kalai, Omer Paneth, and Amit Sahai. Protecting obfuscation against algebraic attacks. Cryptology ePrint Archive, Report 2013/631, 2013. `http://eprint.iacr.org/2013/631`.

[BKL13] Ahto Buldas, Andres Kroonmaa, and Risto Laanoja. *Keyless Signatures' Infrastructure: How to Build Global Distributed Hash-Trees*, pages 313–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[Bla93] Matt Blaze. A cryptographic file system for UNIX. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 9–16, New York, NY, USA, 1993. ACM.

[Bla94] Matt Blaze. Key management in an encrypting file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, Berkeley, CA, USA, 1994. USENIX Association.

[BLOO11] Amos Beimel, Yehuda Lindell, Eran Omri, and Ilan Orlov. 1/p-secure multiparty computation without honest majority and the best of both worlds. In *Annual Cryptology Conference*, pages 277–296. Springer, 2011.

[BMC$^+$15] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy*, pages 104–121, San Jose, CA, USA, may 2015. IEEE Computer Society Press.

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

[BP15] Alex Biryukov and Ivan Pustogarov. *Proof-of-Work as Anonymous Micropayment: Rewarding a Tor Relay*, pages 445–455. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[BR13] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. Cryptology ePrint Archive, Report 2013/563, 2013. `http://eprint.iacr.org/2013/563`.

[BR14] Zvika Brakerski and Guy N. Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *TCC '14*, pages 1–25. Springer, 2014.

[CHK04] Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In *EUROCRYPT*, pages 207–222, 2004.

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

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 364–369. ACM, 1986.

[coi] coinmarketcap.com. `https://coinmarketcap.com/`.

[cpp] cpp-ethereum. cpp-ethereum. Available at `https://github.com/ethereum/cpp-ethereum`.

[CS04] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J. Comput.*, 33(1):167–226, January 2004.

[DHWW11] Alan M Dunn, Owen S Hofmann, Brent Waters, and Emmett Witchel. Cloaking malware with the trusted platform module. In *USENIX Security Symposium*, 2011.

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

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

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

[FG16] David Fitzpatrick and Drew Griffin. Cyber-extortion losses skyrocket, says FBI. Available at http://money.cnn.com/2016/04/15/technology/ransomware-cyber-security/, 2016.

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

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

[GGHW13] Sanjam Garg, Craig Gentry, Shai Halevi, and Daniel Wichs. On the implausibility of differing-inputs obfuscation and extractable witness encryption with auxiliary input. Cryptology ePrint Archive, Report 2013/860, 2013. http://eprint.iacr.org/2013/860.

[GGM13] Christina Garman, Matthew Green, and Ian Miers. Decentralized anonymous credentials. In *NDSS '13*, volume 2013, 2013.

[GGSW13] Sanjam Garg, Craig Gentry, Amit Sahai, and Brent Waters. Witness encryption and its applications. Cryptology ePrint Archive, Report 2013/258, 2013. http://eprint.iacr.org/2013/258.

[GK10] S Dov Gordon and Jonathan Katz. Partial fairness in secure two-party computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 157–176. Springer, 2010.

[GKP$^+$13a] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, , and Nickolai Zeldovich. How to run turing machines on encrypted data. Cryptology ePrint Archive, Report 2013/229, 2013. http://eprint.iacr.org/2013/229.

[GKP$^+$13b] Shafi Goldwasser, Yael Tauman Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. *How to Run Turing Machines on Encrypted Data*, pages 536–553. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[GMMP12] Michael R Grimaila, Justin Myers, Robert F Mills, and Gilbert Peterson. Design and analysis of a dynamically configured log-based distributed security event detection methodology. *The Journal of Defense Modeling and Simulation*, 9(3):219–241, 2012.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

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

[GRFJ14] Mainak Ghosh, Miles Richardson, Bryan Ford, and Rob Jansen. A TorPath to TorCoin: Proof-of-Bandwidth Altcoins for Compensating Relays. In *7th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2014)*, Amsterdam, Netherlands, July 2014.

[Gua] Guardtime. Guardtime. Available at https://guardtime.com/.

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

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

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

[JMH15] Florian Jacob, Jens Mittag, and Hannes Hartenstein. A Security Analysis of the Emerging P2P-Based Personal Cloud Platform MaidSafe. In *TrustCom/BigDataSE/ISPA*, pages 1403–1410. IEEE, 2015.

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

[Kil06] Eike Kiltz. Chosen-ciphertext security from tag-based encryption. In Shai Halevi and Tal Rabin, editors, *TCC '06*, volume 3876, pages 581–600. Springer, 2006.

[KRS$^+$03] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies*, FAST '03, pages 29–42, Berkeley, CA, USA, 2003. USENIX Association.

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

[Las] 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`.

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

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

[MRY04] Philip MacKenzie, Michael K. Reiter, and Ke Yang. Alternatives to non-malleability: Definitions, constructions, and applications. In Moni Naor, editor, *TCC '04*, volume 2951, pages 171–190. Springer, 2004.

[Nak12] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. 2012.

[nam16] Namecoin. `https://namecoin.org/`, November 2016.

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

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

[Pin03] Benny Pinkas. Fair secure two-party computation. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 87–105. Springer, 2003.

[PPW13] Tobias Pulls, Roel Peeters, and Karel Wouters. Distributed privacy-preserving transparency logging. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 83–94, New York, NY, USA, 2013. ACM.

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

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, New York, NY, USA, 2011. ACM.

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

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

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

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

[SK98] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 4–4, Berkeley, CA, USA, 1998. USENIX Association.

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

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

[The] The TrueCrypt Project. Truecrypt. Available at http://truecrypt.sourceforge.net/.

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

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

[TS15] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. Cryptology ePrint Archive, Report 2015/464, 2015. http://eprint.iacr.org/2015/464.

[Twe] TweetNaCl. Tweetnacl. Available at https://tweetnacl.cr.yp.to.

[TZL$^+$17] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE Euro S&P*, 2017.

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

[VWGP12] Jo Vliegen, Karel Wouters, Christian Grahn, and Tobias Pulls. Hardware Strengthening a Distributed Logging Scheme. In *Digital System Design*. IEEE, 2012.

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

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.

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

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

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

A Security and Correctness definitions for Custodian-Contract Interaction

We now present a formal simulation-based definition of security for a custodian-contract interaction protocol.

The **Real** *experiment.* The real-world experiment is parameterized on $(\Sigma, \mathcal{F}, \mathcal{U}, q, q_{\mathsf{forge}})$. Let $\mathcal{U}$ be an adversary playing the role of the user. First, run $\mathsf{Setup}_{\mathcal{F}}$ to create the secret S. Next, initialize $\mathsf{z}_0 \leftarrow \varepsilon$, $\mathsf{STenc}_{i-1} \leftarrow \varepsilon$ and $\mathcal{T}_{\mathsf{Pub}} \leftarrow \varepsilon$. For $i = 1$ to q, the adversary $\mathcal{U}$ may issue the following queries:

Query to the contract functionality. When $\mathcal{U}$ submits (C_i, P_{i-1}, U_i), run $\mathsf{ExecuteContract}_{\mathcal{F}}(C_i, U_i, \mathsf{z}_{i-1})$. Append the public output to $\mathcal{T}_{\mathsf{Pub}}$, record the new state z_i, and return all outputs to $\mathcal{U}$.

Query to the custodian functionality. When $\mathcal{U}$ submits $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, [\pi_i, O_i])$, run $\mathsf{ExecuteCustodian}_{\mathcal{F}}$ on the provided inputs and secret S. Return the output to $\mathcal{U}$.

Query the forgery oracle. $\mathcal{U}$ may query the forgery oracle F up to q_{forge} times through the protocol. When $\mathcal{U}$ querys F on $[C_f \| C_{f-1} \| D_f \| P_f]$, generate a validating authenticator π_f.

Let $\mathcal{U}_q$ be the adversary's output following the q^{th} oracle query. Let $\mathcal{T}_{\mathsf{Pub}}$ be the transcript of all outputs produced by the contract. The output of the **Real** experiment is defined as $(\mathcal{U}_q \| \mathcal{T}_{\mathsf{Pub}})$.

The **Ideal** *experiment.* This experiment is parameterized on $(\mathcal{F}, \mathcal{S}, q)$. Let $\mathcal{S}$ be an ideal-world adversary that conducts the following experiment. First, run $\mathsf{Setup}_{\mathcal{F}}$ to obtain S. Set $\mathsf{ST}_i \leftarrow \varepsilon$. For $i = 1$ to q, the ideal-world adversary $\mathcal{S}$ may query an oracle $\mathcal{O}^{\mathcal{F},S,F}$. When $\mathcal{S}$ submits Input_i, U_i to this oracle, the oracle performs the following steps:

1. Compute $D_{i+1} \leftarrow \mathsf{DoContract}(P_{i-1}, U_i, \mathsf{z}_i)$ and append the public output to $\mathcal{T}_{\mathsf{Pub}}$.
2. Uniformly sample $r \in \{0,1\}^{\ell}$.
3. Compute $(\mathsf{Output}_i \| P_i, \mathsf{ST}_i) \leftarrow \mathsf{NextStep}(S, i, \mathsf{Input}_i, \mathsf{ST}_{i-1}, D_i, r))$.

Let $\mathcal{S}_q$ be the ideal-world adversary's output following the q^{th} oracle query. The output of the **Ideal** experiment is defined as $(\mathcal{S}_q \| \mathcal{T}_{\mathsf{Pub}})$, where P_q is the public output of the contract. We are now ready to formalize the definition of simulation security for a Custodian-Contract scheme. Our definition of security requires that for all p.p.t. real-world adversaries $\mathcal{U}$ there must exist an ideal-world adversary $\mathcal{S}^{\mathcal{O}^{\mathcal{F},S,F}(\cdot)}$ such that the distribution of outputs produced $\mathcal{U}$ at the conclusion of the real-world experiment is indistinguishable from that of the ideal-world adversary $\mathcal{S}$ at the conclusion of the ideal-world experiment.

A.1 Third Party Privacy

Our main security definition (Definition A.1) considers the integrity of CCS computations in the face of an adversarial User. However, a useful Custodian-Contract scheme must also address the privacy of a honest User against curious third parties, who may eavesdrop on (public) consensus transactions in order to recover non-public User inputs or outputs. We address this concern via a separate security experiment that models an honest Custodian, an honest User and an *honest-but-curious* Contract who acts as the observer, and may read any protocol messages sent to or from the Contract. Intuitively we wish to ensure that for each of our protocols, the curious Contract cannot learn the input Input_i. We address this property via a real/ideal world definition. In the *real-world experiment*, the adversary $\mathcal{A}$ is provided with an oracle that allows her to request the User run the computation on a given set of inputs. $\mathcal{A}$ receives each input and output from the $\mathsf{ExecuteContract}$ call that results from this computation. In the *ideal-world experiment*, we require the existence of a simulator $\mathcal{S}$ that, on input a valid query from $\mathcal{A}$ (*i.e.,* a query that would otherwise produce a valid output from $\mathsf{ExecuteUser}$), makes the necessary query to $\mathsf{ExecuteContract}$. The key in this definition is that $\mathcal{S}$ does receive $\mathcal{A}$'s input (or any previous output). Our definition requires that no p.p.t $\mathcal{A}$ can distinguish the two experiments.

The third-party Privacy Real experiment ($\mathbf{Real}_{\mathsf{TPP}}$). The experiment proceeds as follows. For $i = 1$ to q, the adversary $\mathcal{A}$ may query an oracle on tuples of the form $(\mathsf{Input}_i, U_i, C')$. In response the oracle samples random coins r_i and returns $C_i \leftarrow \mathsf{ExecuteUser}_{\mathcal{F}}(\mathsf{Input}_i, C_{i-1}, r_i)$. The oracle then completes the protocol of §2.1 by running $\mathsf{ExecuteContract}$ and $\mathsf{ExecuteCustodian}$ on the resulting outputs, and finally provides $\mathcal{A}$ with all input/output values sent to the Custodian and the Contract:

$$(C_i, U_i, P_{i-1}, \mathsf{z}_{i-1}, O_i, \pi_i, \mathsf{STenc}_i, \mathsf{STenc}_{i-1}, \mathsf{Output}_i, P_i)$$

The third-party Privacy Ideal experiment ($\mathbf{Ideal}_{\mathsf{TPP}}$). This experiment is parameterized by a probabilistic simulator algorithm $\mathcal{S}^{\mathcal{F}}$ that does not have access to the computing secrets S, s. It operates identically to the **Real** experiment except that *in addition* to running the protocol as described above, the oracle also runs $\mathcal{S}^{\mathcal{F}}$ with no user inputs to produce an output C'_i, then calls a second instance of the Contract oracle to produce (O_i, π_i). The oracle provides $\mathcal{A}$ with the tuple:

$$(C'_i, U_i, P_{i-1}, \mathsf{z}_{i-1}, O_i, \pi_i, \mathsf{STenc}_i, \mathsf{STenc}_{i-1}, \mathsf{Output}_i, P_i)$$

In both cases the adversary $\mathcal{A}$ outputs a bit b.

Definition A.1 (Third party privacy) *We say that a Custodian-Contract scheme* Σ_{ccs} *is* third party private *for functionality* $\mathcal{F}$ *if* $\forall$ *p.p.t. real-world adversarial users* $\mathcal{A}$ *and* $\forall$ *non-negative integers* q, q_{forge}*, there exists a p.p.t. simulator* $\mathcal{S}^{\mathcal{F}}$ *such that* $\mathbf{Pr}\left[\mathbf{Real}_{\mathsf{TPP}}(\Sigma_{\mathsf{ccs}}, \mathcal{F}, \mathcal{A}, q)\right] - \mathbf{Pr}\left[\mathbf{Ideal}_{\mathsf{TPP}}(\mathcal{F}, \mathcal{S}^{\mathcal{F}}, q)\right] \leq \nu(\lambda)$.

Remark. We note that this experiment provides privacy for the user inputs because in the ideal-world experiment, the simulator $\mathcal{S}$ does not receive the User input Input_i or random coins specified by the User.

B Proof of Theorem 3.1

We now present a proof of Theorem 3.1. We briefly recall our two experiments, **Real** and **Ideal** from Appendix A. In **Real**, the protocol is run as described in Section 2.2, with $\mathcal{A}$ allowed to make up to q_{forge}

queries to the forgery oracle F. This oracle takes as input $O_f = [C_f \| C_{f-1} \| D_f \| P_f]$ from the adversary and outputs the authenticator π_f (note that we assume it is impossible for $\mathcal{A}^F$ to forge an authenticator idependantly). In the **Ideal** experiment there exists an ideal oracle that runs the program functionality and maintains its own state. Additionally, in the ideal experiment we give $\mathcal{S}$ access to up to q_{forge} calls to Single Step Oracles. After each run of the real ideal functionality, the ideal oracle outputs an encrypted checkpoint that can only be used by the single step oracles. These oracles can run exactly one step of the function $\mathsf{NextStep}$ from a supplied start point and then can never run anything again.

We prove that there exists a simulator $\mathcal{S}$ such that no p.p.t. adversary $\mathcal{A}^F$ can distinguish the distribution of outputs from the **Real** and **Ideal** experiment with non-negligible probability. To prove this statement, we first describe the operation of the simulator $\mathcal{S}$. We then proceed to demonstrate that if there exists a p.p.t $\mathcal{A}^F$ that distinguishes the output of the experiments, then one or more of the following are true: (1) there exists an adversary that breaks the EU-CMA security of the authenticator, (2) an attack on the binding property of the commitment scheme, or (3) a distinguisher for the pseudorandom function PRF. We will now describe the operation of the simulator $\mathcal{S}$.

The operation of $\mathcal{S}$. The simulator $\mathcal{S}$ interacts with the real-world adversary $\mathcal{A}^F$ using the interface described in the real-world experiment. The simulator also has oracle access to the ideal functionality (which cannot be rewound) as well as an authenticator oracle Π that will produce a valid authenticator π for chosen messages. Finally, the simulator has oracle access to a forgery oracle F and q_{forge} Single Step Oracles, as described above. $\mathcal{S}$ maintains three tables, T_{con}, T_{cus}, and T_{forge}. Each query $\mathcal{A}^F$ makes to the various oracles is stored in the appropriate table. Queries to $\mathcal{O}^{\text{Contract}}$ are stored in T_{con}, queries to $\mathcal{O}^{\text{Custodian}}$ are stored in T_{cus}, and queries to F are stored in T_{forge}. For example the i^{th} row of T_{con} contains

$$(C_i, U_i \pi_i, O_i)$$

Similarly, T_{cus} stores all the queries and responses $\mathcal{A}$ makes and receives to $\mathcal{O}^{\text{Custodian}}$. The i^{th} row contains:

$$(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, \pi_i, O_i, \mathsf{Output}_i, P_i, \mathsf{STenc}_{i+1})$$

and T_{forge} contains the following in the i^{th} row:

$$(\pi_i, O_i)$$

All tables are initialized to be empty. $\mathcal{S}$ receives and processes queries from $\mathcal{A}^F$ as follows. When $\mathcal{A}$ attempts to the query the contract oracle on C_i, U_i, the simulator does the following:

1. $\mathcal{S}$ calls $\mathcal{O}^{\text{DoContract}}$ to generate D_i
2. $\mathcal{S}$ finds C_{i-1} in T_{con} and generates $O_i = [C_i \| C_{i-1} \| D_i \| P_i]$. $\mathcal{S}$ then queries the signing oracle Π on O_i and receives π_i. $\mathcal{S}$ adds the entry (C_i, π_i, O_i) to T_{con}.
3. $\mathcal{S}$ returns (π_i, O_i) to $\mathcal{A}$.

When $\mathcal{A}$ attempts to query the custodian oracle on $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, \pi_i, O_i)$, the simulator does the following:

1. $\mathcal{S}$ checks T_{cus} for any previous entries matching the query. If there are any, $\mathcal{S}$ returns $(\mathsf{STenc}_{i+1}, \mathsf{Output}_i)$ from that table entry.
2. $\mathcal{S}$ checks if the π_i is in T_{forge}. If it is, $\mathcal{S}$ runs $(\mathsf{STenc}_{i+1}, \mathsf{Output}_i) \leftarrow \textbf{SingleStep}(\mathsf{STenc}_i, \mathsf{Input}_i)$. $\mathcal{S}$ halts and outputs $(\mathsf{STenc}_{i+1}, \mathsf{Output}_i)$.
3. $\mathcal{S}$ looks up π_i, O_i in the ith row of T_{con} and makes sure they match some entry. Otherwise, $\mathcal{S}$ halts and returns $\perp$.
4. $\mathcal{S}$ parses $O_i = [C_i \| C_{i-1} \| D_i \| P_i]$ and checks that $C_i = \mathsf{Commit}(\mathsf{Input}_i, C_{i-1}; r_i)$. Otherwise, $\mathcal{S}$ halts and returns $\perp$.
5. If $\mathsf{STenc}_i = \varepsilon$, $\mathcal{S}$ checks that $C_{i-1} = \varepsilon$. If it does not, $\mathcal{S}$ halts and returns $\perp$.
6. $\mathcal{S}$ checks that STenc_i is in row $i-1$ of T_{cus}. Otherwise, $\mathcal{S}$ halts and returns $\perp$
7. $\mathcal{S}$ calls $\mathcal{O}^{\text{DoWork}}$ on Input_i and receives Output_i.

8. $\mathcal{S}$ samples $\mathsf{STenc}_{i+1} \xleftarrow{\$} \{0,1\}^\ell$ and appends $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, O_i, \mathsf{Output}_i, \mathsf{STenc}_{i+1})$ to T_{con}.
9. $\mathcal{S}$ returns $\mathsf{Output}_i, \mathsf{STenc}_{i+1}$ to $\mathcal{A}$.

When $\mathcal{A}$ attempts to query F on $(C_f, C_{f-1}, D_f \| P_f)$, the simulator does the following:

1. $\mathcal{S}$ simulates the forgery oracle F and generates π_f
2. $\mathcal{S}$ appends (π_f, O_f) to T_{forge}

Let $\mathcal{A}$ be an adversarial user that succeeds in distinguishing the output of the **Ideal** and **Real** experiments. We now show that such an adversary violates one of our assumptions above. The proof proceeds via a series of hybrids, where the first hybrid (**Game 0**) is identically distributed to the **Real** experiment, and the final hybrid represents the **Ideal** experiment instantiated with the simulator $\mathcal{S}$ described above. For notational convenience, let $\mathbf{Adv}\,[\,\mathbf{Game\ i}\,]$ be $\mathcal{A}$'s advantage in distinguishing the output of **Game i** from **Game 0**, *i.e.,* the **Real** distribution.

Game 0. In this hybrid, $\mathcal{S}$ responds to queries as in the **Real** experiment.

Game 1. In this hybrid, we modify $\mathcal{S}$ to abort and output $\mathsf{Event}_{\mathsf{forge}}$ in the event that $\mathcal{A}$ queries $\mathcal{O}^{\text{Custodian}}$ on input $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, [\pi_i, O_i])$ for any valid authentication tag (π_i, O_i) that was not supplied by a previous oracle call to $\mathcal{S}$. We note that if $\mathcal{A}$ succeeds in producing this event with non-negligible probability, then we obtain a trivial attack on the $\mathsf{SUF\text{-}AUTH}$ of the contract authenticator. Since by assumption the probability of such an event is negligible, we bound $\mathbf{Adv}\,[\,\mathbf{Game\ 1}\,] \leq \nu_1(\lambda)$.

Game 2. In this hybrid we modify the operation of $\mathcal{S}$ such that it aborts and outputs $\mathsf{Event}_{\mathsf{collision}}$ in the event that $\mathcal{A}$ queries $\mathcal{O}^{\text{Custodian}}$ twice with commitment inputs $C_i = C_j$ and two *valid* distinct commitment openings $(\mathsf{Input}_i, C_i, r_i) \neq (\mathsf{Input}_j, C_j, r_j)$. By Lemma B.1 we prove that this event occurs with at most negligible probability, hence $\mathbf{Adv}\,[\,\mathbf{Game\ 2}\,] - \mathbf{Adv}\,[\,\mathbf{Game\ 1}\,] \leq \nu_2(\lambda)$.

Game 3. In this hybrid we modify $\mathcal{S}$ to generate each state encryption key k_i by sampling uniformly at random from $\{0,1\}^\ell$. Recall that in the preceding hybrid, each k_i is generated as $\mathsf{PRF}_S(\text{“}\mathsf{enc}\text{”}\|C_i)$. Note as well that if the Simulator has not aborted, then each C_i is unique. Thus if there exists an attacker that distinguishes **Game 3** from **Game 2** with non-negligible advantage, then we can trivially construct a distinguisher that distinguishes the output of PRF_S (evaluated on unique inputs) from that of a random function. Thus under the assumption that PRF is pseudorandom we have that $\mathbf{Adv}\,[\,\mathbf{Game\ 3} - \mathbf{Game\ 2}\,] \leq \nu_3(\lambda)$.

Game 4. In this hybrid we modify the operation of $\mathcal{S}$ such that it aborts and outputs $\mathsf{Event}_{\mathsf{auth}}$ in the event that $\mathcal{A}$ queries $\mathcal{O}^{\text{Custodian}}$ on valid input STenc_{i-1}, where STenc_i was not supplied as an output from a previous call to this oracle (excepting the special case where $\mathsf{STenc}_{i-1} = C_i = \varepsilon$). By Lemma B.2 we have that $\mathbf{Adv}\,[\,\mathbf{Game\ 4} - \mathbf{Game\ 3}\,] \leq \nu_4(\lambda)$.

Game 5. In this hybrid we modify the operation of $\mathcal{S}$ so that each ciphertext STenc_i is replaced with a random string of equivalent length. By Lemma B.3, $\mathbf{Adv}\,[\,\mathbf{Game\ 5} - \mathbf{Game\ 4}\,] \leq \nu_5(\lambda)$.

Game 6. In this hybrid we modify the operation of $\mathcal{S}$ such that when $\mathcal{A}$ queries the custodian repeatedly on the same tuple $(\mathsf{STenc}_i, \mathsf{Input}_i, r_i, [\pi_i, O_i])$, all queries subsequent to the first query return the same result as the first query. Recall that by definition the $\mathsf{ExecuteCustodian}$ algorithm is deterministic; thus, repeated queries on the same input will always produce the same output. Hence by definition $\mathbf{Adv}\,[\,\mathbf{Game\ 6} - \mathbf{Game\ 5}\,] = 0$.

The advantage gained by the adversary in distinguishing between the **Real** experiment and the **Ideal** experiment is bounded by the negligible function $\nu_1(\lambda) + \cdots + \nu_5(\lambda) + 0$. Thus by the hybrid lemma, the distributions of **Game 6** and **Game 0** are computationally indistinguishable.

We now conclude with the following Lemmas.

Lemma B.1 (Input Commitments are Unique) *If the commitment scheme Σ_{com} is binding, then* $\mathbf{Adv}\,[\,\mathbf{Game\ 2}\,] - \mathbf{Adv}\,[\,\mathbf{Game\ 1}\,] \le \nu_2(\lambda)$.

Proof. Let $(C_0, \ldots, C_k)$ be the sequence of valid commitments provided to the Custodian after the k^{th} successful invocation (*i.e.,* those commitments for which $\mathcal{A}$ has provided a correct opening tuple). Recall that for $i \ge 0$ each commitment is computed as $C_i = \mathsf{Commit}(\mathsf{Input}_i \| C_{i-1}; r_i)$ with the special case that the initial commitment $C_0 = \varepsilon$. For the abort condition of **Game 2** to occur, there must be at least two duplicate commitments in the list above, and argue that this event will occur with at most negligible probability if the commitment scheme is binding. We proceed using induction.

Let us consider only a prefix of the list that consists of the commitments $(C_0, \ldots, C_j)$ for some $j \ge 0$. Clearly there can be no duplicate commitments if $j \le 1$. This would imply that there is either a single commitment, or that $C_1 = C_0 = \varepsilon$ which is not possible by our definition of the commitment scheme. Now let us consider the inductive case. Let $j > 1$ be an integer such that $(C_0, \ldots, C_{j-1})$ contains no duplicate commitments, but $C_j = C_l$ for some $0 < l < j$.[19] Then we have two cases. In the first case if $(\mathsf{Input}_j, C_{j-1}, r_j)$ $= (\mathsf{Input}_l, C_{l-1}, r_l)$ then we have $C_{j-1} = C_{l-1}$. However this violates our initial assumption that there are no duplicates in the set $(C_0, \ldots, C_{j-1})$. This leaves only the second case where the two input tuples are not equal, yet the resulting commitments are identical. If $\mathcal{A}$ causes this to occur with non-negligible probability, we can use $\mathcal{A}$ to construct an attacker that violates the binding property of the commitment scheme with identical probability. Thus, under the assumption that commitments are binding, we have that $\mathbf{Adv}\,[\,\mathbf{Game\ 2} - \mathbf{Game\ 1}\,] \le \nu_2(\lambda)$. □

Lemma B.2 (Unforgeability of State Ciphertexts) *If the authenticated encryption scheme Σ_{AE} satisfies the AE definition, then* $\mathbf{Adv}\,[\,\mathbf{Game\ 3} - \mathbf{Game\ 2}\,] \le \nu_3(\lambda)$.

Proof. If $\mathbf{Adv}\,[\,\mathbf{Game\ 3} - \mathbf{Game\ 2}\,]$ is non-negligible, this implies that $\mathcal{A}$ induces $\mathsf{Event}_{\mathsf{auth}}$ with non-negligible probability. We show that such an adversary can be used to construct an adversary $\mathcal{B}$ that has a non-negligible advantage against the authenticity definition of Rogaway [Rog02]. Let q be the maximum number of Custodian oracle calls that can be made by $\mathcal{A}$. $\mathcal{B}$ runs $\mathcal{A}$ internally. It first samples an integer $j \in [0, q)$ and runs the protocol of **Game 3**. For each query $i \ne j$, runs the protocol as usual. At the j^{th} query, $\mathcal{B}$ uses the AE challenge oracle to produce the output ciphertext STenc_j using the correct input and a 0 message (of the same length) as the challenge pair. If $\mathcal{A}$ submits a forged value $\mathsf{STenc}_{j+1} \ne \mathsf{STenc}_j$ then $\mathcal{B}$ sends this value to the decryption oracle to learn the bit b. If $\mathcal{A}$ succeeds with probability ϵ, then $\mathcal{B}$ will succeed with advantage ϵ/q. Since $\mathcal{A}$ can forge the state ciphertext with at most negligible probability, then $\mathbf{Adv}\,[\,\mathbf{Game\ 3} - \mathbf{Game\ 2}\,] \le \nu_3(\lambda)$. □

Lemma B.3 (Indistinguishability of State Ciphertexts) *If the authenticated encryption scheme Σ_{AE} satisfies the AE definition, then* $\mathbf{Adv}\,[\,\mathbf{Game\ 5} - \mathbf{Game\ 4}\,] \le \nu_5(\lambda)$.

Proof. We assume Σ_{AE} schemes satisfy the random ciphertext model of security: formally $\mathbf{Adv}^{\mathrm{priv}}_{\Pi}(\mathcal{A}) = \mathbf{Pr}\left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\mathcal{E}_K(\cdot,\cdot)} = 1 \right] - \mathbf{Pr}\left[\mathcal{A}^{\$(\cdot,\cdot)} = 1 \right]$. Let k be a random encryption key. We show that if $\mathcal{A}$ can distinguish the distribution of **Game 4**, where STenc_i is a valid encryption of a state value, and **Game 5** where STenc_i is a random string of the same language, we can construct an adversary $\mathcal{B}$ such that $\mathbf{Adv}^{\mathrm{priv}}_{\Pi}(\mathcal{B}) > \nu_5(\lambda)/q$. $\mathcal{B}$ samples a value $j \in [1, q)$ and runs the protocol of **Game 4**. To answer $\mathcal{A}$'s j^{th} query, $\mathcal{B}$ queries the encryption oracle and returns the result as STenc_j. Note that if the result is a correct encryption of the message, the distribution is as in **Game 4**. If the result is a random string, then the distribution is as in **Game 5**. If $\mathcal{A}$ distinguishes the two distributions with non-negligible advantage, then $\mathcal{B}$ uses $\mathcal{A}$'s output to obtain a non-negligible $\mathbf{Adv}^{\mathrm{priv}}_{\Pi}(\mathcal{B})$. Since we assume that the AE scheme is secure, this produces a contradiction. Thus $\mathbf{Adv}\,[\,\mathbf{Game\ 5} - \mathbf{Game\ 4}\,] \le \nu_5(\lambda)$. □

[19]We ignore the case where $C_l = C_0 = \varepsilon$, since this cannot occur by definition.

C A Multiparty Public Key Encryption Scheme

In this section we sketch a construction for the multi-party public key encryption scheme we use in §4.4. While the techniques used to construct this scheme are not novel, we include them for completeness. First, we briefly define multi-party public key encryption.

C.1 Multiparty Public Key Encryption (MPKE)

Let $\mathcal{M}$ be a message space. A multiparty public key encryption (MPKE) scheme consists of three possibly probabilistic algorithms $(\mathsf{PKMGen}, \mathsf{PKMEnc}, \mathsf{PKMDec})$ with the following description:

$\mathsf{PKMGen}(1^\lambda) \rightarrow (pk, sk)$. On input a security parameter, the key generation algorithm outputs a public and secret key.

$\mathsf{PKMEnc}(pk_1, \ldots, pk_n, m) \rightarrow CT$. On input a set of n public keys and a message m, the encryption algorithm outputs a single ciphertext CT.

$\mathsf{PKMDec}(pk, sk, CT) \rightarrow m'$. On input a single public and secret key as well as a ciphertext CT, outputs a decrypted message or the distinguished error symbol $\perp$.

Correctness. Intuitively, a MPKE scheme is *correct* if $\forall m \in \mathcal{M}$, integers $n > 0$ and $0 < j \leq n$, and key vectors $(pk_1, \ldots, pk_n)$ where each $(pk_i, sk_i) \in \mathsf{PKMGen}(1^\lambda)$, the following relation holds with probability $\geq 1 - \nu(\lambda)$ over the random coins of the experiment:

$$\mathsf{PKMDec}(pk_j, sk_j, \mathsf{PKMEnc}(pk_1, \ldots, pk_n, m)) = m.$$

Security. Let the random variable $\mathsf{IND\text{-}MPKE\text{-}CCA}_b^n(\Pi, \mathcal{A}, \lambda)$ where $b \in \{0,1\}$, integer $n > 0$, $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\lambda \in \mathbb{N}$ denote the result of the following probabilistic experiment:

```
$\mathsf{IND\text{-}MPKE\text{-}CCA}_b^n(\Pi, \mathcal{A}, \lambda)$
    $(pk_1, sk_1) \leftarrow \mathsf{PKMGen}(1^\lambda), \ldots, (pk_n, sk_n) \leftarrow \mathsf{PKMGen}(1^\lambda)$
    $(m_0, m_1, z) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{dec}}(\cdot,\cdot)}(pk_1, \ldots, pk_n)$ s.t. $m_0, m_1 \in \mathcal{M}$
    $y \leftarrow \mathsf{PKMEnc}(pk_1, \ldots, pk_n, m_b)$
    $B \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{dec}}(\cdot,\cdot)}(y, z)$
    Output $B$
```

We define $\mathcal{O}_{\mathsf{dec}}(i, CT)$ to be a decryption oracle that, on input an integer i and a ciphertext CT outputs $\mathsf{PKMDec}(pk_i, sk_i, CT)$ if $CT \neq y$, and otherwise outputs $\perp$.

An MPKE scheme Π is $\mathsf{IND\text{-}MPKE\text{-}CCA}$-secure for keysets of size n if $\forall$ p.p.t. algorithms $\mathcal{A}$ the following two ensembles are computationally indistinguishable:

$$\left\{\mathsf{IND\text{-}MPKE\text{-}CCA}_0^n(\Pi, \mathcal{A}, \lambda)\right\}_\lambda \overset{c}{\approx} \left\{\mathsf{IND\text{-}MPKE\text{-}CCA}_1^n(\Pi, \mathcal{A}, \lambda)\right\}_\lambda$$

Remark. We note that the decryption oracle $\mathcal{O}_{\mathsf{dec}}$ operates over a *single* public key pk_i and does not take as input the remaining public keys. Our definition requires that decryption oracle operates correctly even without access to this additional information.

C.2 An MPKE scheme from TBE

We now describe one exemplary construction of an MPKE scheme. The key ingredient in our construction is a Tag-Based Encryption (TBE) scheme [MRY04]. TBE is a form of public-key encryption in which the encryption and decryption algorithm take an additional input called a *tag*, drawn from an exponentially-sized tag space. TBE schemes are known from a variety of underlying assumptions in the standard model, including DDH in cyclic groups [CS04] and well-studied bilinear assumptions [Kil06].[20]

Tag-based Encryption. A TBE scheme consists of three (possibly probabilistic) algorithms (TBEKG, $\mathsf{TBEEnc}, \mathsf{TBEDec}$). Key generation operates identically to a public key encryption scheme. On input a public key, a message m and a tag T, TBEEnc produces a ciphertext CT. Similarly, on input a public and secret key, a ciphertext and a tag T, TBEDec outputs a message m' or $\perp$.

We require a TBE that provides *selective tag* CCA security ($\mathsf{IND\text{-}sTBE\text{-}CCA}$). Intuitively this definition is a variant of the classical $\mathsf{IND\text{-}CCA2}$ definition, in which the adversary has access to a decryption oracle that, on input a tag T and a ciphertext CT, provides the decryption of CT with tag T under the following restriction: the attacker may not query the decryption oracle on the specific tag T^* that was used to formulate the challenge ciphertext. In the selective tag model the attacker selects the tag T^* but must do so at the start of the experiment, prior to receiving any public keys or parameters. See [Kil06, §3.2] for a formal definition of the experiment.

The MPKE construction. Using TBE as a building block, we now describe our main construction of MPKE. This approach employs the TBE-to-PKE construction independently proposed by MacKenzie *et al.* [MRY04] and Boneh *et al.* [CHK04]. Let $\mathcal{M}$ be the message space of the underlying TBE scheme, and let $\mathcal{T}$ be the scheme's exponentially-sized tag space. We additionally require a $\mathsf{SUF\text{-}CMA}$ one-time signature scheme $\mathsf{OTS} = (\mathsf{OTSKG}, \mathsf{OTSSign}, \mathsf{OTSVerify})$ where the public keyspace for the OTS is included in (or can be mapped to) $\mathcal{T}$.

$\mathsf{PKMGen}(1^\lambda)$. Output $(pk, sk) \leftarrow \mathsf{TBEKG}(1^\lambda)$.

$\mathsf{PKMEnc}(pk_1, \ldots, pk_n, m) \rightarrow CT$. First, sample a keypair for the one-time signature $(svk, ssk) \leftarrow \mathsf{OTSKG}()$. Let p represent a mapping between public keys and ciphertexts. Now for $i = 1$ to n compute $T_i \leftarrow svk$ and $C_i \leftarrow \mathsf{TBEEnc}(pk_i, T_i, m)$ and $\sigma \leftarrow \mathsf{OTSSign}(svk, ssk, (p, C_1, \ldots, C_n))$. Finally, output $CT \leftarrow (p, C_1, \ldots, C_n, \sigma)$.

$\mathsf{PKMDec}(pk, sk, CT) \rightarrow m'$. First parse CT as $(p, C_1, \ldots, C_n, \sigma)$. Now verify the signature σ using $\mathsf{OTSVerify}(svk, (p, C_1, \ldots, C_n), \sigma)$. If this check outputs 0, return $\perp$. Otherwise use p to identify the ciphertext C_i corresponding to pk (if none is found, output $\perp$). Set $T_i \leftarrow svk$ and output $m' \leftarrow \mathsf{TBEDec}(pk, sk, T_i, C_i)$.

Security. We leave correctness as an exercise for the reader, and proceed with a brief sketch of the security argument for the above scheme, which proceeds nearly identically to the arguments of [MRY04, CHK04].

Proof sketch. Let $\mathcal{A}$ be an adversary that succeeds with non-negligible advantage in the $\mathsf{IND\text{-}MPKE\text{-}CCA}$ experiment against the MPKE scheme. We construct a second adversary $\mathcal{B}$ that succeeds with non-negligible advantage in the $\mathsf{IND\text{-}sTBE\text{-}CCA}$ experiment for the TBE scheme. At the start of the experiment, $\mathcal{B}$ generates a keypair (svk^*, ssk^*) for the OTS, and selects $T^* = svk$. When $\mathcal{B}$ receives pk from the TBE challenger, it selects a random integer $j \in [n]$ and sets $pk_j \leftarrow pk$. For $k = 1$ to n, $k \neq j$ it computes $(pk_k, sk_k) \leftarrow \mathsf{PKMGen}(1^\lambda)$ and retains the secret key. $\mathcal{B}$ now runs $\mathcal{A}$ on input $(pk_1, \ldots, pk_n)$. Whenever $\mathcal{A}$ queries the decryption oracle on (CT, i) where $svk = T^*$ $\mathcal{B}$ returns $\perp$. Otherwise $\mathcal{B}$ first verifies the attached signature and outputs $\perp$ on failure. If the signature verifies and $i = j$, $\mathcal{B}$ queries the TBE decryption oracle on C_i and returns the result. If $i \neq j$ then $\mathcal{B}$ computes the decryption using the known secret key. To produce the challenge ciphertext y, $\mathcal{B}$ transmits (m_0, m_1) to the TBE challenge oracle and sets C_j to be the result. $\mathcal{B}$ samples a random bit b' and for $k = 1$ to n (and $k \neq j$) computes $C_k \leftarrow \mathsf{TBEEnc}(pk_i, T^*, m_b)$ and then signs the resulting vector $(p, C_1, \ldots, C_n)$ using ssk to produce y. When $\mathcal{A}$ outputs a guess B, $\mathcal{B}$ outputs B as its guess.

[20]Moreover, if TBE schemes are not available, then Identity-Based Encryption schemes can be used for TBE. This implies a much wider range of assumptions from which these schemes may be constructed.

We argue that with probability $1/2$ $\mathcal{A}$ will select b' such that $b' = b$, and thus the distribution of y will be identical to that of the real protocol. We further argue that in this case all outputs from the decryption oracle $\mathcal{O}_{\mathsf{dec}}$ are distributed identically to the real protocol, which the exception of the case where $svk = T^*$ and $CT \neq y$. In that one case, however, it is easy to show that $\mathcal{B}$'s simulation will differ from the real experiment if and only if $\mathcal{A}$ is able to produce a new pair (σ, m) not signed by $\mathcal{B}$ such that this pair validates using svk. We argue that if $\mathcal{A}$ succeeds in this with non-negligible probability, that would imply a successful adversary against the $\mathsf{SUF\text{-}CMA}$ property of the OTS. Therefore this condition must occur with at most negligible probability. Hence if $\mathcal{A}$ succeeds in the $\mathsf{IND\text{-}MPKE\text{-}CCA}$ experiment with advantage ϵ then $\mathcal{B}$ succeeds in the $\mathsf{IND\text{-}sTBE\text{-}CCA}$ experiment with advantage at least $\epsilon/2 - \nu(\lambda)$. $\square$