The Ring of Gyges: Using Smart Contracts for Crime

Ari Juels
Jacobs Institute, Cornell Tech
juels@cornell.edu

Ahmed Kosba
Univ. of Maryland
akosba@cs.umd.edu

Elaine Shi
Cornell Univ.
runting@gmail.com

> The Ring of Gyges is a mythical magical artifact mentioned by the philosopher Plato in Book 2 of his *Republic*. It granted its owner the power to become invisible at will.
> –Wikipedia, "Ring of Gyges"
>
> "[On wearing the ring,] no man would keep his hands off what was not his own when he could safely take what he liked out of the market, or go into houses and lie with anyone at his pleasure, or kill or release from prison whom he would, and in all respects be like a God among men."
> –Plato, *The Republic*, Book 2 (2.360b) (trans. Benjamin Jowett)

***Abstract*—Thanks to their anonymity (pseudonymity) and lack of trusted intermediaries, cryptocurrencies such as Bitcoin have created or stimulated growth in many businesses and communities. Unfortunately, some are criminal, e.g., money laundering, marketplaces for illicit goods, and ransomware.**

Next-generation cryptocurrencies such as Ethereum will include rich scripting languages in support of *smart contracts*, programs that autonomously intermediate transactions. We illuminate the extent to which these new cryptocurrencies, by enabling criminal activities to be conducted anonymously and with minimal trust assumptions, may fuel new criminal ecosystems. Specifically, we show how what we call *criminal smart contracts* (CSCs) can facilitate leakage of confidential information, theft of cryptographic keys, and various real-world crimes (murder, arson, terrorism).

We show significantly that CSCs for leakage of secrets are efficiently realizable in existing scripting languages such as that in Ethereum. We show that CSCs for key theft can be achieved using cryptographic primitives, such as Succinct Non-interactive ARguments of Knowledge (SNARKs), that are already expressible in these languages and for which efficient supporting language extensions are anticipated. We demonstrate similarly that authenticated data feeds, another anticipated feature of smart contract systems, can facilitate CSCs for real-world crimes.

Our results illuminate the scope of possible abuses in next-generation cryptocurrencies. They highlight the urgency of creating policy and technical safeguards and thereby realizing the great promise of smart contracts for beneficial goals.

1. Introduction

Cryptocurrencies such as Bitcoin remove the need for trusted third parties from basic monetary transactions and offer anonymous (more accurately, pseudonymous) transactions between individuals. While attractive to some, these features have a dark side. Bitcoin has stimulated the growth of ransomware [7], money laundering [39], and illicit commerce, as exemplified by the notorious Silk Road [31].

New cryptocurrencies such as Ethereum (as well as systems such as Counterparty [45] and SmartContract [1]) will offer even richer functionality than Bitcoin. They support *smart contracts*, a generic term denoting programs written in Turing-complete cryptocurrency scripting languages. In a fully distributed system such as Ethereum, smart contracts enable general fair exchange (atomic swaps) without a trusted third party, and thus can effectively guarantee payment for committed crimes. It is thus to be expected that such smart contract systems will stimulate new forms of crime.

We refer to smart contracts that facilitate crimes in distributed smart contract systems as *criminal smart contracts* (CSCs). An example of a CSC is a smart contract for (private-)key theft. Such a CSC might pay a reward for delivery of a target key sk, such as a certificate authority's private digital signature key. (For confidentiality, sk might be encrypted under the public key of the contract creator.)

The key questions we explore in this paper are: *Could CSCs enable a wider range of significant new crimes than earlier cryptocurrencies (Bitcoin)?*; *How practical will such new crimes be?*; and *What key advantages do CSCs provide to criminals compared with other means?* Exploring these questions is essential to identifying threats and exploring countermeasures.

CSC Challenges. Would-be criminals face two basic challenges in the construction of CSCs. First, it is not immediately obvious whether a CSC is at all feasible for a given crime, such as key theft. This is because it is challenging to ensure that a CSC is what we call *commission-fair*, meaning that its execution guarantees *both* commission of a crime and commensurate payment for the perpetrator of the crime or *neither*. Fair exchange alone is not sufficient to ensure commission-fairness: We show how CSC constructions leveraging fair exchange in a natural way still allow a

party to a CSC to cheat.

Second, even if a CSC can in principle be constructed, given the restricted languages in existing smart contract systems (such as Ethereum), it is not immediately clear that the CSC can be made *practical*. By this we mean that the CSC can be executed without unduly burdensome computational effort, which in some smart contract systems (e.g., Ethereum) would also mean unacceptably high fees levied against the CSC.

This paper. We show in this paper that it is indeed possible in envisioned decentralized smart contract systems to construct commission-fair CSCs for three types of crime: leakage / sale of secret documents, theft of private keys, and a very broad class of physical-world crimes (murder, arson, etc.) that we refer to as "calling-card" crimes. (That this last type of CSC is possible is somewhat surprising, and relies, as we explain, on the anticipated deployment of authorities attesting to real-world facts.) We formally prove the correctness of our CSC constructions. We also show experimentally in Ethereum that the CSCs we explore are practical—or will be with planned language extensions.

By highlighting what CSCs are practical, as well as their fragility, our work illuminates the need for and potentially methods for constructing defenses, which we briefly discuss. Criminal activity committed under the guise of anonymity has posed a major impediment to adoption for Bitcoin. Yet there has been little discussion of criminal contracts in public forums on cryptocurrency [15] and the launch of Ethereum took place in July 2015. By recognizing and protecting against CSCs early in their lifecycle, we hope to see the great promise of distributed smart contract systems fully realized.

1.1. Smart contracts: the good and bad

Decentralized smart contracts have many beneficial uses, including the realization of a rich variety of new financial instruments. Informally, a smart contract in a decentralized system such as Ethereum may be thought of as an autonomously executing piece of code whose inputs and outputs can include money. (We give more formalism below.) As Bitcoin does for transactions, in a decentralized smart contract system, *the consensus system enforces autonomous execution of contracts*; no one entity or small set of entities can interfere with the execution of a contract. As contracts are self-enforcing, they eliminate the need for trusted intermediaries or reputation systems to reduce transactional risk. Decentralized smart contracts offer several advantages over traditional cryptocurrencies such as Bitcoin:

- *Fair exchange* between mutually distrustful parties with *rich contract rules expressible in a programmable logic*; this feature prevents parties from cheating by aborting an exchange protocol, yet removes the need for physical rendezvous and (potentially cheating) third-party intermediaries;
- *Minimized interaction* between parties also for a *rich set of contracts expressible in a programmable logic*, reducing opportunities for unwanted monitoring and tracking;
- *Enriched transactions with external state* by allowing as input *authenticated data feeds* (attestations) provided by brokers on physical and other events outside the smart-contract system, e.g., stock tickers, weather reports, etc.

Unfortunately, for all of their benefit, these properties have a dark side, potentially facilitating crime because:

- *Fair exchange* enables transactions between mutually distrustful criminal parties, eliminating the need for today's fragile reputation systems and/or potentially cheating or law-enforcement-infiltrated third-party intermediaries [40], [54].
- *Minimized interaction* renders illegal activities harder for law enforcement to monitor. In some cases, as for the key-theft and calling-card CSCs we present, a criminal to set up a contract and walk away, allowing it to execute autonomously with no further interaction.
- *Enriched transactions with external state* significantly broaden the scope of possible CSCs to take in, e.g., physical crimes (terrorism, arson, murder, etc.).

As decentralized smart contract systems typically inherit the anonymity (pseudonymity) of Bitcoin, they offer similar secrecy for criminal activities. Broadly speaking, therefore, there is a risk that the capabilities enabled by decentralized smart contract systems will *enable new underground ecosystems and communities*.

1.2. Example CSC: Key theft

To exemplify the power of decentralized smart contracts for CSC creation, as well as the challenge of constructing commission-fair CSCs, we consider a key-theft CSC as an example.

Example 1a (Key compromise contract). *Contractor $\mathcal{C}$ posts a request for theft and delivery of the signing key $\mathsf{sk}_{\mathcal{V}}$ of a victim certificate authority (CA) "Certs 'R Us." $\mathcal{C}$ offers a reward* $\mathsf{reward} *to a perpetrator $\mathcal{P}$ for delivering the Certs 'R Us private key $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{C}$.*

In the Bitcoin ecosystem, there is no automated mechanism to enforce this contract. To ensure fair exchange of the key and reward, a necessary condition for commission-fairness (but not a sufficient one), $\mathcal{C}$ and $\mathcal{P}$ would need to use a trusted third party or communicate directly, raising the risk to both parties of being cheated (by the third party) or discovered by law enforcement authorities (if the third party is monitored or infiltrated or one of $\mathcal{C}$ or $\mathcal{P}$ is a mole). They could rely on a reputation system, but such systems are often infiltrated by law enforcement authorities [54].

By using a decentralized smart contract, Example 1a could instead be self-enforcing and thus require no trusted intermediary. We explore the following CSC Contract in detail later in the paper:

Example 1b (Key compromise CSC). *$\mathcal{C}$ generates a private / public key pair $(\mathsf{sk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{C}})$ and initializes* Contract *with*

public keys $\mathsf{pk}_{\mathcal{C}}$ and $\mathsf{pk}_{\mathcal{V}}$ (the Certs 'R Us public key). Upon publication and execution Contract awaits input from a claimed perpetrator $\mathcal{P}$ of a pair (ct, π), where π is a zero-knowledge proof that $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_{\mathcal{C}}}[sk_{\mathcal{V}}]$ is well-formed. The zero-knowledge proof π can be implemented as a SNARK [19] while applying appropriate transformations to lift its security [43]. Contract then verifies π and upon success sends a reward of \$reward to $\mathcal{P}$. The contractor $\mathcal{C}$ can then download and decrypt ct to obtain the compromised key $\mathsf{sk}_{\mathcal{V}}$.

This CSC implements a fair exchange between $\mathcal{C}$ and $\mathcal{P}$. Specifically, the contract pays a reward to $\mathcal{P}$ *if and only if* $\mathcal{P}$ delivers a valid key (as proven by π). This CSC, however, is *not commission-fair*. The CA Certs 'R Us can neutralize the contract by preemptively revoking its own certificate and then itself claiming $\mathcal{C}$'s reward \$reward!

As noted, a major thrust of this paper is showing how, for key cases of interest, it is possible for criminals to construct commission-fair CSCs. Additionally, we show that these CSCs can be efficiently realized using existing cryptocurrency tools or features currently envisioned for cryptocurrencies. For example, we show that the proof π in the key compromise CSC can be constructed and verified efficiently using SNARKs [19] (with appropriate transformations to lift its security [43]).

1.3. Threat Model and Security Guarantees

Threat model. Roughly speaking, we adopt the following thread model.

- *Blockchain: trusted for correctness but not privacy.* We assume that the blockchain will always correctly store data and perform computation, and will always remain available. However, the blockchain exposes all of its internal states to the public, and retains no private data.
- *Arbitrarily malicious contractual parties.* We assume that contractual parties are mutually distrustful, and they act solely to maximize their own benefits. In particular, not only can they deviate arbitrarily from the prescribed protocol, they can also abort from the protocol prematurely.
- *Network influence of the adversary.* We assume that messages in between the blockchain and parties are guaranteed to be delivered within bounded delay. However, an adversary can arbitrarily reorder these messages. In particular, as we show in Section 5 in the construction of a key-theft contract, this enables a rushing attack. We assume that communication channels in between the parties can be unreliable – and an adversary can drop or reorder messages between contractual parties.

The formal model we adopt (see Section 3) formally captures all of the above aspects of the threat model.

Twofold meanings of security. For a CSC to be secure, the meaning is twofold:

- *Securing emulation of the ideal functionality.* First, we show that the real-world protocol securely emulates some desired ideal functionality in the standard Universally Composable (UC) simulation paradigm [26] adopted in the cryptography literature, against arbitrarily malicious contractual counterparties as well as possible network adversaries.
  It is worth noting that our protocols are also secure against aborting adversaries, e.g., a contractual party may attempt to abort without paying the other party. As is well-known by the cryptography community, fairness in the presence of aborts is in general impossible in standard models of distributed computation [32]. But as shown by several recent works, by leveraging a blockchain that is correct, available, and aware of the progression of time, blockchain-based protocols can ensure a certain notion of financial fairness against aborting parties [16], [20], [43]. In particular, aborting behavior can be made evident by the elapse of time, in which case the blockchain may enforce that the aborting party would lose its deposit to the honest parties.
- *Securing ideal functionality itself.* Second, merely proving a UC-style secure emulation does not imply the security of a crimnal smart contract. As we show in Section 5 and in Appendix C, our naive-key theft contract securely emulates a naive (and flawed) key-theft ideal functionality. The problem is that even specifying the correct ideal functionality itself is often challenging! We would like our ideal functionality to satisfy a notion called *commission fairness* as mentioned earlier. Interestingly, we stress that commission fairness is a property of the ideal functionality. We do not give a universal formal definition of commission fairness since its semantic definition is application dependent.

Contributions

Ours is one of the first papers exploring smart contracts with a decentralized cryptocurrency backend. While our focus is on preventing evil, the techniques we propose can equally well underpin constructive contracts. We explore both techniques for structuring CSCs and the use of cutting-edge cryptographic tools, e.g., Succinct Non-interactive ARguments of Knowledge (SNARKs), in CSCs. Like the design of benign smart contracts, CSC construction requires a careful combination of cryptography with commission-fair design [34].

In summary, our contributions are:

- *Criminal smart contracts:* We initiate the study of CSCs as enabled by Turing-complete scripting languages and authenticated data feeds in next-generation cryptocurrencies. We explore CSCs for three different types of crimes: leakage of secrets in Section 4 (e.g., pre-release Hollywood films), key compromise / theft (of, e.g., a CA signing key) in Section 5, and "calling-card" crimes, such as assassination, that use authenticated data feeds in Section 6. We explore the challenges involved in crafting such criminal contracts and demonstrate (anticipate) new techniques to resist neutralization and achieve commission-fairness.
- *Formal security:* We model the smart contract (executed by the blockchain) as a functionality that is *trusted for*

correctness, but not for privacy. In particular, the contract exposes its internal state to all parties. We design protocols in this contract-hybrid world, and prove security under a standard cryptographic simulation paradigm. Our modeling and notational approach follows from the formal blockchain model proposed by Kosba et al. [43].
We give an overview of this formal model and notational system in Section 3. More model details about the model can be found in the recent work by Kosba et al. [43] or in Appendix A.

- *Proof of concept:* To demonstrate that even sophisticated CSC are realistic, we report (in their respective sections) on implementation of the CSCs we explore. Our CSC for leakage of secrets is *efficiently realizable today* in existing smart contract languages (e.g., that of Ethereum). Those for key theft and "calling-card" crimes rely respectively for efficiency and realizability on features currently envisioned by the cryptocurrency community. They too, however, are within practical reach as shown for example for our key-theft CSC, which relies on zk-SNARKs. Our experiments show that verification—the most important function, as it is performed by all full nodes—requires as little as 9.9 msec on a 288-byte proof (with execution on an Amazon EC2 r3.2xlarge instance with 2.5 GHz processors).

We also briefly discuss in Section 7 how maturing technologies, such as hardware roots of trust (e.g., Intel SGX [41]) and program obfuscation can enrich the space of possible CSCs—as they can, of course, beneficial smart contracts.

2. Background and Related Work

We briefly review the cryptocurrency backdrop for CSCs and research on the use of digital cash in crime.

2.1. Blockchains and smart contracts

Emerging decentralized cryptocurrencies [52], [60] rely on a novel blockchain technology, where miners reach consensus, not only about *data*, but also about *computation*. Loosely speaking, the Bitcoin blockchain (i.e., miners) verify transactions and store a global *ledger*, which may be modeled as a piece of public memory whose integrity relies on correct execution of the underlying distributed consensus protocol. Bitcoin offers rudimentary support for a limited range of programmable logic to be executed by the blockchain. Its scripting language is restrictive, however, and difficult to use, as demonstrated by previous efforts at building smart contract-like applications atop Bitcoin [8], [16], [20], [46], [53].

When the computation performed by the blockchain (i.e., miners) is generalized to arbitrary Turing-complete logic, we obtain a more powerful, general-purpose smart contract system. The first embodiment of such a decentralized smart contract system is the soon-to-be-launched Ethereum [60]. Hobbyists and companies are already building atop or forking off Ethereum to develop various smart contract applications such as security and derivatives trading [45], prediction

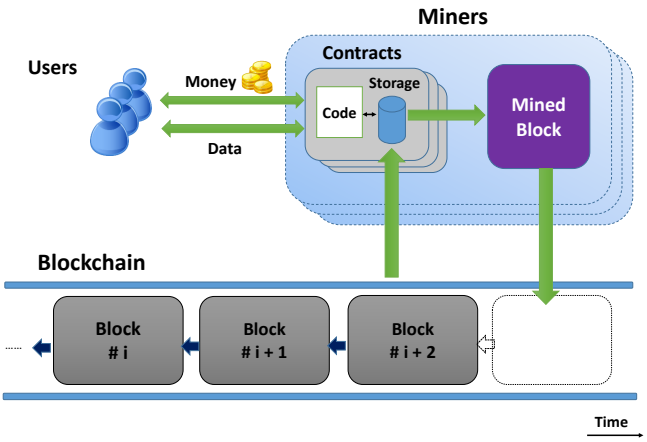

Figure 1: **Schematic of a decentralized cryptocurrency system with smart contracts**, as illustrated by Delmolino et al. [34]. A smart contract's state is stored on the public blockchain. A smart contract program is executed by a network of miners who reach consensus on the outcome of the execution, and update the contract's state on the blockchain accordingly. Users can send money or data to a contract; or receive money or data from a contract.

markets [6], supply chain provenance [12], and crowd fund raising [2].

Figure 1 shows the high-level architecture of a smart contract system instantiated over a decentralized cryptocurrency such as Bitcoin or Ethereum. When the underlying consensus protocol employed the cryptocurrency is secure, the majority of the miners (as measured by computational or other resource) are assumed to correctly execute the contract's programmable logic.

Gas. Realistic instantiations of decentralized smart contract systems rely on *gas* to protect miners against denial-of-service attacks (e.g., running an unbounded contract). Gas is a form of transaction fee that is, roughly speaking, proportional to the runtime of a contract.

In this paper, although we do not explicitly express gas in our smart contract notation, we attempt to factor program logic away from the contract as an optimization when possible, to keep gas and thus transactional fees low. For example, some of the contracts we propose involve program logic executed on the users' side, with no loss in security.

2.2. Digital cash and crime

Anonymous e-cash was introduced in 1982 in a seminal paper by David Chaum [29]. Its dangers were brought to light when Naccache and von Solms noted that anonymous currency would render "perfect crimes" such as kidnapping untraceable by law enforcement [58]. This observation prompted the design of fair blind signatures or "escrow" for e-cash [24], [59], which enables a trusted third party to link identities and payments. Such linkage is possible in classical e-cash schemes where a user identifies herself

upon withdraw of anonymous cash. Escrow schemes cannot easily support contemporary virtual currency systems such as Bitcoin that do not include withdrawal protocols.

Ransomware has appeared in the wild since the advent of the AIDS Info Disk Trojan in 1989 [17]. Young and Yung observed that better cryptographic design could improve the effectiveness of ransomware [61], and coined the term "cryptovirology" for the general use of cryptography to improve malware. Another major cryptovirological "improvement" to ransomware has been use of Bitcoin, now the most common method of payment for ransomware [44]. One strain of ransomware, CryptoLocker, has purportedly netted hundreds of millions of dollars in ransom [22].

There has been extensive study of the enablement of crime through Bitcoin, such as money laundering [51], Bitcoin theft [49], and illegal transactions such as those performed in the Silk Road marketplace [31]. Meiklejohn et al. [49] note that Bitcoin is pseudonymous and that mixes, mechanisms designed to confer anonymity on Bitcoins, do not operate on large volumes of currency and in general today it is hard for criminals to cash out anonymously in volume.

On the other hand, Ron and Shamir provide evidence that the FBI failed to locate most of the Bitcoin holdings of Dread Pirate Roberts, the operator of the Silk Road, even after seizing the laptop of the alleged user behind the nom de guerre, Ross William Ulbricht [56]. Möser, Böhome, and Breuker [51] find that they cannot successfully deanonymize transactions in two of three mixes under study, suggesting that the "Know-Your-Customer" principle, regulators' main tool in combatting money laundering, may prove difficult to enforce in Bitcoin—and probably by extension in the decentralized smart contract systems, such as Ethereum, that we explore here. In addition, there have been increasingly practical proposals to use NIZK proofs to construct stronger anonymity protection for cryptocurrencies [18], [33], [50]. There is therefore good reason to anticipate criminals' ability to achieve a strong degree of anonymity / pseudonymity, particularly as cryptocurrencies mature.

3. Notation and Formalism

We adopt the formal blockchain model proposed by Kosba et al. [43]. As background, we give a high-level description of this model in this section. We use this model to specify cryptographic protocols in our paper; these protocols encompass criminal smart contracts and corresponding user-side protocols.

Protocols in the smart contract model. The model treats a *contract* as a special party that is **entrusted to enforce correctness but not privacy**, as noted above. (In reality, of course, a contract is enforced by the network.) All messages sent to the contract and its internal state are publicly visible. During the protocol, contract would interact with users by exchanging messages (also referred to as transactions). Money, expressed in the form of account balances, is recorded in the global ledger (on the blockchain). Our contracts can access and update the ledger to implement money transfers between users (as represented by their pseudonymous public keys).

Init:	Set $\mathsf{all} := \{\}$, $T_{\text{end}} :=$ 10/12/2015, $\mathsf{price} := 1$.
Register:	On receiving ($\mathsf{amt}, name) from some party $\mathcal{P}$: Assert $\mathsf{name} \notin \mathsf{all}$ and $\mathsf{amt} \geq \mathsf{price}$. $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] - \mathsf{amt}$. $\mathsf{all} := \mathsf{all} \cup \{\mathsf{name}\}$.
Timer:	If $T > T_{\text{end}}$ and $\mathsf{price} = 1$: set $\mathsf{price} := 10$.

Figure 2: **Warmup: a simple smart contract for domain name registration.** The formal operational semantics of a contract program is supplied in Appendix A.

Example contract program. As a warm-up example, Figure 2 gives a simple smart contract using our notation system. This contract sells domain names. A name is awarded to the first bidder to offer at least price currency units. When a presale time period expires indicated by T_{end}, the price of each domain name is increased from 1 to 10 currency units. (The contract does not handle assignment of domain names.)

3.1. Notational Conventions

We now explain some notational conventions for writing contracts.

- **Currency and ledger.** We use $\mathsf{ledger}[\mathcal{P}]$ to denote party $\mathcal{P}$'s balance in the global ledger. For clarity, variables that begin with a \$ sign denote money, but otherwise behave like ordinary variables.
  Unlike in Ethereum's Serpent language, in our formal notation, when a contract receives some amount from a party $\mathcal{P}$, this is only message transfer, and no currency transfer has taken place at this point. Money transfers *only take effect* when the contract performs operations on the ledger, denoted ledger.
- **Pseudonymity.** Parties can use pseudonyms to obtain better anonymity. In particular, a party can generate arbitrarily many public keys. In our notational system, when we refer to a party $\mathcal{P}$, $\mathcal{P}$ denotes the party's pseudonym. Our contract wrapper (see Appendix A) manages the pseudonym generation and the message signing necessary for establishing an authenticated channel to the contract. These details are abstracted away from the main contract program.
- **Timer.** Time progresses in rounds. At the beginning of each round, the contract's **Timer** function will be invoked. The variable T encodes the current time.
- **Entry points and variable scope.** A contract can have various entry points, each of which is invoked when receiving a corresponding message type. Thus entry points behave like function calls invoked upon receipt of messages.
  All variables are assumed to be globally scoped with the following exception: when an entry point says "Upon receiving a message from *some* party $\mathcal{P}$," this allows

the registration of a new party $\mathcal{P}$. In general, contracts are open to any party who interacts with them. When a message is received from $\mathcal{P}$ (without the keyword "some"), party $\mathcal{P}$ denotes a fixed party – and a well-formed contract has already defined $\mathcal{P}$.

3.2. Formal Modeling of Smart Contracts

The notational system described above is not only designed for convenience, but is also endowed with precise, formal meanings. We briefly describe our formal framework below but relegate details to the Hawk work [43] or Appendix A. This notational system is designed modularly such that the main body of the paper can be understood without drilling into the details of the formalism.

Our notational system is compatible with the Universal Composability framework [26], but specially designed for easy expression of cryptographic protocols in the smart contract model of execution.

Wrappers and programs. In support of such modularization, our formal model adopts a wrapper-based based approach. Wrappers factor out repetitive details related to the decentralized smart contract execution model, as we now explain.

Our protocols are described in the so-called $\mathcal{G}(\mathsf{Contract})$-hybrid world, where $\mathcal{G}(\cdot)$ a contract *wrapper*, and $\mathsf{Contract}$ is called a contract *program*. In this paper, all contracts we write (e.g., Contract $\mathsf{KeyTheft}$) are in the form of contract *programs*. Contract programs describe the user-defined portion of a contract. While contract programs (e.g., $\mathsf{KeyTheft}$) are written for intuitive understanding, they must be combined with the contract wrapper $\mathcal{G}$ to be endowed with formally precise operational semantics.

Specifically, the contract wrapper $\mathcal{G}(\cdot)$ models several elements that underlie decentralized smart contract systems such as Bitcoin [52] and Ethereum [60], including 1) the lapse of *time*, 2) handling of *pseudonyms*, 3) maintenance of a global *ledger* and handling of *currency transfer* requests, and 4) publicly exposing the contract's internal states.

A contract interacts with user programs, and therefore user programs must be specified for the protocol description to be complete. In addition to the $\mathcal{G}(\cdot)$ contract wrapper, using a similar design rationale, our formal model also defines wrappers for *user programs* as well as *ideal functionalities*. We relegate the details of our modeling choices to Appendix A.

4. CSCs for Leakage of Secrets

As a first example of the power of smart contracts, we show how an existing type of criminal contract can be made more robust and functionally enhanced through implementation as a smart contract, and can in fact be practically deployed in a system such as Ethereum.

Among the illicit practices stimulated by Bitcoin is payment-incentivized *leakage*, i.e., public disclosure, of secrets. The recently created web site Darkleaks [3] serves as a decentralized market for crowd-patronized public leakage of a wide variety of secrets, including, "Hollywood movies, trade secrets, government secrets, proprietary source code, industrial designs like medicine or defence, [etc.]." (It may be viewed as a subsidized variant on Wikileaks.)

In this section, using Darkleaks as an example, we highlight the functional limitations of Bitcoin in constructing contracts for leakage of secrets. We then show how smart contracts can remedy these limitations to achieve commission-fairness with high probability.

4.1. Darkleaks

In the Darkleaks system, a contractor $\mathcal{C}$ who wishes to sell a piece of content M partitions it into a sequence of n segments $\{m_i\}_{i=1}^n$. At a time (block height) T_{open} prespecified by $\mathcal{C}$, a randomly selected subset $\Omega \subset [n]$ of k segments is publicly disclosed as a sample to entice donors / purchasers—those who will contribute to the purchase of M for public leakage. When $\mathcal{C}$ determines that donors have collectively paid a sufficient price, $\mathcal{C}$ decrypts the remaining segments for public release. The parameter triple (n, k, T_{open}) is set by $\mathcal{C}$ (where $n = 100$ and $k = 20$ are recommended defaults).

To ensure a fair exchange of M for payment without direct interaction between parties, Darkleaks implements a (clever) protocol on top of the Bitcoin scripting language. The main idea is that for a given segment m_i of M that is not revealed as a sample in Ω, donors make payment to a Bitcoin account a_i with public key is pk_i. The segment m_i is encrypted under a key $\kappa = H(\mathsf{pk}_i)$ (where H = SHA-256). To spend its reward from account a_i, $\mathcal{C}$ is forced by the Bitcoin transaction protocol to disclose pk_i; thus the act of *spending the reward automatically enables the community to decrypt* m_i.

We give further details in Appendix E.1.

Shortcomings and vulnerabilities. The Darkleaks protocol has three major shortcomings / vulnerabilities that appear to stem from fundamental functional limitations of Bitcoin's scripting language when constructing contracts without direct communication between parties. The first two undermine commission-fairness, while the third limits functionality.[1] DarkLeaks has these shortcomings:

1. *Delayed release:* $\mathcal{C}$ can refrain from spending purchasers' / donors' payments and releasing unopened segments of M until after M loses value. E.g., $\mathcal{C}$ could withhold segments of a film until after its release in theaters, of an industrial design until after it is produced, etc.

2. *Selective withholding:* $\mathcal{C}$ can choose to forego payment for select segments and not disclose them. For example, $\mathcal{C}$ could leak and collect payment for all of a leaked film but the last few minutes (which, with high probability, will not

1. That these limitations are fundamental is evidenced by calls for new, time-dependent opcodes. One example is CHECKLOCKTIMEVERIFY; apart from its many legitimate applications, proponents note that it can facilitate secret leakage as in Darkleaks [36].

appear in the sample Ω), significantly diminishing the value of leaked segments.

3. *Public leakage only:* Darkleaks can only serve to leak secrets *publicly*. It does not enable fair exchange for *private leakage*, i.e., for payment in exchange for a secret M encrypted under the public key of a purchaser $\mathcal{P}$.

Additionally, Darkleaks has a vulnerability due to a protocol flaw:

4. *Reward theft:* In the Darkleaks protocol, the Bitcoin private key sk_i corresponding to pk_i is derived from m_i; specifically $\mathsf{sk}_i = \text{SHA-256}(m_i)$. Thus, the source of M (e.g., the owner of a leaked film) can derive sk_i and steal rewards received by $\mathcal{C}$. (Also, when $\mathcal{C}$ claims a reward, a malicious node that receives the transaction can decrypt m_i, compute $\mathsf{sk}_i = \text{SHA-256}(m_i)$, and potentially steal the reward by flooding the network with a competing transaction [37].)

This last problem is easily remedied by generating the set $\{\kappa_i\}_{i=1}^n$ of segment encryption keys pseudorandomly or randomly, which we do in our CSC designs.

Remark: While selective withholding (shortcoming 2. above) is inherent to Darkleaks, there is a similar vulnerability that affects *any* protocol in which a random sample is displayed to entice buyers, including what we now present. It is possible for the contractor to insert a small number of incorrectly encrypted or simply valueless segments into M. With non-negligible probability, these will not appear in the sample Ω. The larger k and n, the smaller the risk of such attack. It is in this sense that revelation of Ω provides only a *weak* guarantee of the global validity of M. A formal analysis is outside the scope of this paper.

4.2. A generic public-leakage CSC

We now present a smart contract that realizes public leakage of secrets using blackbox cryptographic primitives. (We later present efficient realizations.) This contract overcomes limitation 1. of the Darkleaks protocol (delayed release) by enforcing disclosure of M at a pre-specified time T_{end}—or else immediately refunding buyers' money. It addresses limitation 2. (selective withholding) by ensuring that M is revealed in an all-or-nothing manner. (We later explain how to achieve private leakage and overcome limitation 3.)

Again, we consider a setting in which $\mathcal{C}$ aims to sell M for public release after revealing a sample of segments M^*.

Informal protocol description. Informally, the protocol involves the following steps:

- *Create contract.* A seller $\mathcal{C}$ initializes a smart contract with the encryption of a randomly generated *master secret key* msk. The master secret key is used to generate (symmetric) encryption keys for the segments $\{m_i\}_{i=1}^n$. $\mathcal{C}$ provides a cryptographic commitment $c_0 := \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$ of msk to the contract. (To meet the technical requirements of our security proofs, the commitment is an encryption with randomness r_0 under a public key pk created during a trusted setup step.) The master secret key msk can be used to decrypt all leaked segments of M.
- *Upload encrypted data.* For each $i \in [n]$, $\mathcal{C}$ generates encryption key $\kappa_i := \mathsf{PRF}(\mathsf{msk}, i)$, and encrypts the i-th segment as $\mathsf{ct}_i = \mathsf{enc}_{\kappa_i}[m_i]$. $\mathcal{C}$ sends all encrypted segments $\{\mathsf{ct}_i\}_{i\in[n]}$ to the contract (or, for efficiency, provides hashes of copies stored with a storage provider, e.g., a peer-to-peer network). Interested purchasers / donors can download the segments of M, but cannot decrypt them yet.
- *Challenge.* The contract generates a random challenge set $\Omega \subset [n]$, in practice based on the hash of the most recent currency block, or using other well-known randomness sources such as the NIST randomness beacon [10].
- *Response.* $\mathcal{C}$ reveals the set $\{\kappa_i\}_{i\in\Omega}$ to the contract, and gives ZK proofs that the revealed secret keys $\{\kappa_i\}_{i\in\Omega}$ are generated correctly from the msk encrypted as c_0.
- *Collect donations.* During a donation period, potential purchasers / donors can use the revealed secret keys $\{\kappa_i\}_{i\in\Omega}$ to decrypt the corresponding segments. If they like the decrypted segments, they can donate money to the contract as contribution for the leakage.
- *Accept.* If $\mathcal{C}$ determines that enough money has been collected, $\mathcal{C}$ decommits msk for the contract (sends the randomness for the ciphertext along with msk). If the contract verifies the decommitment successfully, all donated money is paid to $\mathcal{C}$. The contract thus enforces a fair exchange of msk for money. (If the contract expires at time T_{end} without release of msk, all donations are refunded.)

The contract. Our proposed smart contract PublicLeaks for implementing this public leakage protocol is given in Figure 3. The corresponding user-side is as explained informally (and inferrable from the contract); thus we defer a formal specification of these protocols to Appendix D, in which we also describe an ideal functionality for the public leakage contract, and formally prove that our protocols securely emulate this ideal functionality.

Formal proof. In a formal proof, PublicLeaks is shown to realize an ideal functionality that ensures commission-fairness assuming all revealed segments are valid—a property enforced with high (although not overwhelming) probability by the **Create** CSC functionality. The security proof is relegated to Appendix D.

4.3. Optimizations and Ethereum implementation

The contract PublicLeaks uses generic cryptographic primitives in a blackbox manner. We now give a practical, optimized version relying on the random oracle model (ROM) that eliminates a trusted setup, and also achieves better efficiency and easy integration with Ethereum [60].

A practical optimization. During contract creation, $\mathcal{C}$ chooses random $\kappa_i \xleftarrow{\$} \{0,1\}^\lambda$ for $i \in [n]$, and computes

$$c_0 := \{H(\kappa_1, 1), \ldots, H(\kappa_n, n)\}.$$

Contract PublicLeaks

Init: Set state := INIT, and donations := {}. Let crs := $\mathsf{KeyGen}_{\text{nizk}}(1^\lambda)$, pk := $\mathsf{KeyGen}_{\text{enc}}(1^\lambda)$ denote hardcoded public parameters generated through a trusted setup.

Create: Upon receiving ("create", c_0, $\{\mathsf{ct}_i\}_{i=1}^n$, T_{end}) from some leaker $\mathcal{C}$:

Set state := CREATED.
Select a random subset $\Omega \subset [n]$ of size k, and send ("challenge", Ω) to $\mathcal{C}$.

Confirm: Upon receiving ("confirm", $\{(\kappa_i, \pi_i)\}_{i\in\Omega}$) from $\mathcal{C}$:

Assert state = CREATED.
Assert that $\forall i \in S$: π_i is a valid NIZK proof (under crs) for the following statement:

$$\exists(\mathsf{msk}, r_0), \text{ s.t. } (c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)) \wedge (\kappa_i = \mathsf{PRF}(\mathsf{msk}, i))$$

Set state := CONFIRMED.

Donate: Upon receiving ("donate", $amt) from some purchaser $\mathcal{P}$:

Assert state = CONFIRMED.
Assert ledger$[\mathcal{P}] \geq$ $amt.
Set ledger$[\mathcal{P}]$:= ledger$[\mathcal{P}]$ − $amt.
donations := donations $\cup \{(\$\mathsf{amt}, \mathcal{P})\}$.

Accept: Upon receiving ("accept", msk, r_0) from $\mathcal{C}$:

Assert state = CONFIRMED
Assert $c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$
ledger$[\mathcal{C}]$:= ledger$[\mathcal{C}]$ + sum(donations)
Send ("leak", msk) to all parties.
Set state := ABORTED.

Timer: If state = CONFIRMED and $T > T_{\text{end}}$: $\forall(\$\mathsf{amt}, \mathcal{P}) \in$ donations: let ledger$[\mathcal{P}]$:= ledger$[\mathcal{P}]$ + $amt. Set state := ABORTED.

Figure 3: A contract PublicLeaks that leaks a secret M to the public in exchange for donations.

The master secret key is simply $\mathsf{msk} := \{\kappa_1, \ldots, \kappa_n\}$, i.e., the set of hash pre-images. As in PrivateLeaks, each segment m_i will still be encrypted as $\mathsf{ct}_i := \mathsf{enc}_\kappa[m_i]$. (For technical reasons—to achieve simulatability in the security proof— here $\mathsf{enc}_\kappa[m_i] = m_i \oplus [H(\kappa_i, 1, \text{"enc"}) \,||\, H(\kappa_i, 2, \text{"enc"}) \ldots, \,||\, H(\kappa_i, z, \text{"enc"})]$ for suitably large z.)

$\mathcal{C}$ submits c_0 to the smart contract. When challenged with the set Ω, $\mathcal{C}$ reveals $\{\kappa_i\}_{i\in\Omega}$ to the contract, which then verifies its correctness by hashing and comparing with c_0. Finally, to accept donations, $\mathcal{C}$ reveals the entire msk.

Although this optimized scheme is asymptotically less efficient than our generic, blackbox construction PublicLeaks— as the master secret key scales linearly in the number of segments n— for typical document set sizes encountered in realistic settings (e.g., $n = 100$, as recommended for Darkleaks), it is more efficient in practice.

Finally, we make a remark on the security proof for the optimized scheme given in the full version of the paper. Our overall proof structure would remain the same for the optimized scheme, under the ROM for H. For schemes under the ROM to be universally composable, each protocol instance needs to instantiate a different random oracle, or the approach of Canetti et al. [28] can be adopted.

Ethereum-based implementation. To demonstrate the feasibility of implementing secret-leakage contracts using currently available technology, we implemented a version of the contract PublicLeaks atop Ethereum [60], using their Serpent contract language [11]. We specify the full implementation in detail in Appendix E.2.

The version we implemented relies on the practical optimizations described above. As a technical matter, Ethereum does not appear at present to support timer-activated functions, so we implemented **Timer** in such a way that purchasers / donors make explicit withdrawals, rather than receiving automatic refunds.

This public leakage Ethereum contract is highly efficient, as it does not require expensive cryptographic operations. It mainly relies on hashing (SHA3-256) for random number generation and for verifying hash commitments. The total number of storage entries (needed for encryption keys) and hashing operations is $O(n)$, where we recall that $n = 100$ is the Darkleaks recommended parameter. Each hash function call in practice takes a few micro-seconds, e.g. **3.92** μ**secs** measured on a core i7 processor.

4.4. Extension: private leakage

As noted above, shortcoming 3. of Darkleaks is its inability to support *private* leakage, in which $\mathcal{C}$ sells a secret exclusively to a purchaser $\mathcal{P}$. In Appendix E.3, we show how PublicLeaks can be modified for this purpose. The basic idea is for $\mathcal{C}$ not to reveal msk directly, but to provide a ciphertext $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_\mathcal{P}}[\mathsf{msk}]$ on msk to the contract for a purchaser $\mathcal{P}$, along with a proof that ct is correctly formed. We describe a blackbox variant whose security can be proven in essentially the same way as PublicLeaks. We also describe a practical variant that variant combines a *verifiable random function* (VRF) of Chaum and Pedersen [30] (for generation of $\{\kappa_i\}_{i=1}^n$) with a *verifiable encryption* (VE) scheme of Camensich and Shoup [25] (to prove correctness of ct). This variant can be deployed today using beta features in Ethereum's Serpent scripting language that support big number arithmetic and is efficient enough for practical use.

5. A Key-Compromise CSC

Example 1b described a CSC that rewards a perpetrator $\mathcal{P}$ for delivering to $\mathcal{C}$ the stolen key $sk_\mathcal{V}$ of a victim $\mathcal{V}$— in this case a certificate authority (CA) with public key $pk_\mathcal{V}$. $\mathcal{C}$ generates a private / public key encryption pair $(\mathsf{sk}_\mathcal{C}, \mathsf{pk}_\mathcal{C})$. The contract accepts as a claim by $\mathcal{P}$ a pair (ct, π). It sends reward $reward to $\mathcal{P}$ if π is a valid proof that $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_\mathcal{C}}[\mathsf{sk}_\mathcal{V}]$ and $\mathsf{sk}_\mathcal{V}$ is the private key corresponding to $\mathsf{pk}_\mathcal{V}$.

Contract KeyTheft-Naive

Init: Set $\mathsf{state} := \text{INIT}$. Let $\mathsf{crs} := \mathsf{KeyGen}_{\text{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Upon receiving ("create", $\$\mathsf{reward}$, $\mathsf{pk}_\mathcal{V}$, T_{end}) from some contractor $\mathcal{C} := (\mathsf{pk}_\mathcal{C}, \ldots)$:

Assert $\mathsf{state} = \text{INIT}$.
Assert $\mathsf{ledger}[\mathcal{C}] \geq \$\mathsf{reward}$.
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] - \$\mathsf{reward}$.
Set $\mathsf{state} := \text{CREATED}$.

Claim: Upon receiving ("claim", ct, π) from some purported perpetrator $\mathcal{P}$:

Assert $\mathsf{state} = \text{CREATED}$.
Assert that π is a valid NIZK proof (under crs) for the following statement:

$$\exists r, \mathsf{sk}_\mathcal{V} \text{ s.t. } \mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_\mathcal{C}, (\mathsf{sk}_\mathcal{V}, \mathcal{P}), r) \text{ and } \mathsf{match}(\mathsf{pk}_\mathcal{V}, \mathsf{sk}_\mathcal{V}) = \mathtt{true}$$

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$.
Set $\mathsf{state} := \text{CLAIMED}$.

Timer: If $\mathsf{state} = \text{CREATED}$ and current time $T > T_{\text{end}}$:
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$
$\mathsf{state} := \text{ABORTED}$

Figure 4: A naïve, flawed key theft contract (lacking commission-fairness)

This form of contract can be used to solicit theft of any type of private key, e.g., the signing key of a CA, the private key for a SSL/TLS certificate, a PGP private key, etc. (Similar contracts can be applied to solicit abuse, but not full compromise of a private key, e.g., to obtain a forged certificate.)

Figure 4 shows the contract of Example 1b in our notation for smart contracts. We let crs here denote a common reference string for a NIZK scheme and $\mathsf{match}(\mathsf{pk}_\mathcal{V}, \mathsf{sk}_\mathcal{V})$ denote an algorithm that verifies whether $\mathsf{sk}_\mathcal{V}$ is the corresponding private key for some public key $\mathsf{pk}_\mathcal{V}$ in a target public-key cryptosystem.

As noted above and as we now discuss, this CSC is *not* commission-fair. Thus we refer to it as KeyTheft-Naive. We use KeyTheft-Naive as a helpful starting point for motivating and understanding the construction of a commission-fair contract proposed later, called KeyTheft.

5.1. Flaws in KeyTheft-Naive

The contract KeyTheft-Naive fails to achieve commission-fairness due to two shortcomings.

Revoke-and-claim attack. The more serious vulnerability is an attack in which the CA $\mathcal{V}$ revokes the key $\mathsf{sk}_\mathcal{V}$ and then submits it for payment. The CA then not only negates the value of the contract but actually profits from it. This *revoke-and-claim* attack demonstrates that KeyTheft-Naive is not commission-fair in the sense of ensuring for payment the delivery of a *usable* private key sk_V. Related to the revoke-and-claim attack, there is another problem: The target and state of contract KeyTheft-Naive are publicly visible. $\mathcal{V}$ can thus learn whether it is the target of KeyTheft-Naive. $\mathcal{V}$ also learns of the existence of a successful claim—and thus whether $\mathsf{sk}_\mathcal{V}$ has been stolen. $\mathcal{V}$ can thus take informed defensive action against the contract. For example, as key revocation is expensive and time-consuming, $\mathcal{V}$ may choose not to perform a revoke-and-claim attack when KeyTheft-Naive is posted, but instead wait until a successful claim occurs and then revoke its key.

Rushing attack. Another attack is a rushing attack, where a corrupted contractor deprives the pepertrator of its reward. Specifically, notice that in our formalism in Section 3 and in Appendix A, the contract wrapper $\mathcal{G}$ notifies the adversary of any message it receives without delay. Therefore, a corrupted contractor can immediately decrypt and learn $\mathsf{sk}_\mathcal{V}$, and construct another valid claim to compete with the perpetrator $\mathcal{P}$. In real-life decentralized cryptocurrencies, the contractor's claim and the perpetrator's claim will arrive in the same round, and the order in which they are processed by the winning miner will determine who gets the reward. To model potential attacks where an adversary bribes miners, in our formalism we allow the adversary to submit a permutation on all transactions arriving in the same round.

We now show how to modify KeyTheft-Naive to prevent revoke-and-claim and rushing attacks and thus achieve commission-fairness. We also propose techniques for hiding the target and state of KeyTheft-Naive.

5.2. Thwarting Attacks

Thwarting the revoke-and-claim attack. In a revoke-and-claim attack against KeyTheft-Naive, $\mathcal{V}$ preemptively revokes its public key $\mathsf{pk}_\mathcal{V}$ and replaces it with a fresh one $\mathsf{pk}'_\mathcal{V}$. As noted above, the victim can then play the role of perpetrator $\mathcal{P}$, submit $\mathsf{sk}_\mathcal{V}$ to the contract and itself claim the reward. The result is that $\mathcal{C}$ pays $\$\mathsf{reward}$ to $\mathcal{V}$ and obtains a stale key.

We address this problem by adding to the contract a feature called *reward truncation*, whereby the contract accepts evidence of revocation Π_{revoke}.

This evidence Π_{revoke} can assume any of several forms. It can be an Online Certificate Status Protocol (OCSP) response indicating that $\mathsf{pk}_\mathcal{V}$ is no longer valid, a new certificate for $\mathcal{V}$ that was unknown at the time of contract creation (and thus not stored in Contract), or a certificate revocation list (CRL) containing the certificate with $\mathsf{pk}_\mathcal{V}$.

Although $\mathcal{C}$ could submit Π_{revoke}, to minimize interaction by $\mathcal{C}$, KeyTheft could instead provide a reward $\$\mathsf{smallreward}$ to a third-party submitter. The reward can be small, as Π_{revoke} would be easy for an ordinary user to obtain.

The contract then provides a reward based on the interval of time over which the key $\mathsf{sk}_\mathcal{V}$ remains valid. Let T_{claim} denote the time at which the key $\mathsf{sk}_\mathcal{V}$ is provided and T_{end} be an expiration time for the contract (which must not exceed the expiration of the certificate containing the targeted

key). Let T_{revoke} be the time at which Π_{revoke} is presented ($T_{\text{revoke}} = \infty$ if no revocation happens prior to T_{end}). Then the contract assigns to $\mathcal{P}$ a reward of $f(\mathsf{reward}, t)$, where $t = \min(T_{\text{end}}, T_{\text{revoke}}) - T_{\text{claim}}$, and thus $t \leq T_{\text{end}} - T_{\text{claim}}$.

We do not explore choices of f here. We note, however, that given that a CA key $\mathsf{sk}_{\mathcal{V}}$ can be used to forge certificates for rapid use in, e.g., malware or falsified software updates, much of its value can be realized in a short interval of time which we denote by δ. (A slant toward up-front realization of the value of exploits is common in general [21].) A suitable choice of reward function should be front-loaded and rapidly decaying. A natural, simple choice with this property is

$$f(\$\mathsf{reward}, t) = \begin{cases} 0 & : t < \delta \\ \$\mathsf{reward}(1 - ae^{-b(t-\delta)}) & : t \geq \delta \end{cases}$$

for $a < 1/2$ and some positive real value b. Note that a majority of the reward is paid provided that $t \geq \delta$.

Thwarting the rushing attack. To thwart the rushing attack, our idea is to separate the claim into two phases. In the first phase, the perpetrator expresses an intent to claim by submitting a commitment of the real claim message. The pepertrator then waits for the next round, and then opens the commitment and reveals the claim message. Due to technical subtleties in the proof, the commitment must be *adaptively secure* – in the proof, the simulator must be able to simulate a commitment without knowing the string s being committed to, and later, be able to explain the commitment to any string s. In real-life decentralized cryptocurrencies, the perpetrator can potentially wait multiple block intervals before opening the commitment, to have higher confidence that the blockchain cannot be reversed. In our formalism, one round can potentially correspond to one or more block intervals.

Figure 5 gives a key theft contract KeyTheft that thwarts the revoke-and-claim attack and the rushing attack.

Formal proof. We relegate a formal proof of security for KeyTheft to Appendix C.

5.3. Limiting target and state exposure

To limit target and state exposure, we propose two possible enhancements to KeyTheft. The first is a *multi-target* contract, in which key theft is requested for any one of a set of multiple victims. The second is use of what we call *cover claims*, false claims that conceal any true claim and for which a small reward is paid. Our implementation of KeyTheft, described below, is a multi-target contract, as this technique provides both partial target and partial state concealment.

Multi-target contract. A multi-target contract solicits the private key of any of m potential victims $\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_m$. There are many settings in which the private keys of different victims are of similar value. For example, a multi-target contract KeyTheft could offer a reward for the private key $\mathsf{sk}_{\mathcal{V}}$ of *any* CA able to issue SSL/TLS certificates trusted by, e.g., Internet Explorer (of which there are more than 650 [38]).

A challenge here is that because the contract state is public, the contract must be able to verify the submission of a valid claim (private key) $\mathsf{sk}_{\mathcal{V}_i}$ *without knowing which key was furnished, i.e., without learning* i. We show in our implementation that the construction of such proofs as zk-SNARKs is practical. (Note that determination of i by $\mathcal{C}$ is quite efficient: $\mathcal{C}$ decrypts $\mathsf{sk}_{\mathcal{V}_i}$, generates $\mathsf{pk}_{\mathcal{V}_i}$, and identifies the corresponding victim.)

Cover claims. As the state of a contract is publicly visible, a victim $\mathcal{V}$ learns whether or not a successful claim has been submitted to KeyTheft-Naive. This is particularly problematic in the case of single-target contracts.

Rather than sending the NIZK proof π with ct, we can instead delay submission of π (and payment of the reward) until T_{end}. (That is, Claim takes as input ("claim", ct).) This approach conceals the validity of ct. Note that even without π, $\mathcal{C}$ can still make use of ct.

A contract that supports such concealment can also support an idea that we refer to as *cover claims*. A cover claim is an *invalid* claim of the form ("claim", ct), i.e., one in which ct is not a valid encryption of $\mathsf{sk}_{\mathcal{V}}$. Cover claims may be submitted by $\mathcal{C}$ to conceal the true state of the contract. Alternatively, so that $\mathcal{C}$ need not interact with the contract after its creation, the contract can parcel out a small reward at time T_{end} to third parties that submit cover claims.

1-Target	*#threads*	**RSA-2048**	**ECDSA_P256**
Key Gen.[$\mathcal{C}$]	1	418.27 sec	926.308 sec
	4	187.49 sec	421.05 sec
Eval. Key		0.78GB	1.80 GB
Ver. Key		17.29 KB	15.6 KB
Prove[$\mathcal{P}$]	1	133.06 sec	325.73 sec
	4	**55.30 sec**	**150.80 sec**
Proof		288 B	288 B
Verification [Contract]		**0.0102 sec**	**0.0099 sec**

500-Target	*#threads*	**RSA-2048**	**ECDSA_P256**
Key Gen.[$\mathcal{C}$]	1	419.93 sec	934.89 sec
	4	187.88 sec	329.39 sec
Eval. Key		0.79 GB	1.81 GB
Ver. Key		1.14 MB	330.42 KB
Prove[$\mathcal{P}$]	1	132.98 sec	325.73 sec
	4	**68.67 sec**	**149.19 sec**
Proof		288 B	288 B
Verification [Contract]		**0.0316 sec**	**0.0159 sec**

TABLE 1: Performance of the key-compromise zk-SNARK circuit for **Claim** in the case of a 1-target and 500-target contracts. [.] refers to the entity performing the computational work.

5.4. Implementation

We rely on zk-SNARKs for efficient realization of the protocols above. zk-SNARKs are zero-knowledge proofs of knowledge that are succinct and very efficient to verify. zk-SNARKs have weaker security than what is needed in UC-style simulation proofs. We therefore we use a generic transformation described in the Hawk work [43] to lift

Contract KeyTheft

Init: Set state := INIT. Let crs := $\mathsf{KeyGen}_{\text{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Same as in Contract KeyTheft-Naive (Figure 4), except that an additional parameter ΔT is additionally submitted by $\mathcal{C}$.

Intent: Upon receiving ("intent", cm) from some purported perpetrator $\mathcal{P}$:

Assert state = CREATED
Assert that $\mathcal{P}$ has not sent "intent" earlier
Store $\mathsf{cm}, \mathcal{P}$

Claim: Upon receiving ("claim", ct, π, r) from $\mathcal{P}$:

Assert state = CREATED
Assert $\mathcal{P}$ submitted ("intent", cm) earlier such that $\mathsf{cm} = \mathsf{comm}(\mathsf{ct}||\pi, r)$.

Continue in the same manner as in contract KeyTheft-Naive, except that the ledger update $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$ does not take place immediately.

Revoke: On receive ("revoke", Π_{revoke}) from some $\mathcal{R}$:

Assert Π_{revoke} is valid, and state $\neq$ ABORTED.
$\mathsf{ledger}[\mathcal{R}] := \mathsf{ledger}[\mathcal{R}] + \$\mathsf{smallreward}$.
If state = CLAIMED:
Let t := (time elapsed since successful **Claim**).
Let $\mathcal{P}$:= (successful claimer).
$\mathsf{reward}_{\mathcal{P}} := f(\$\mathsf{reward}, t)$.
$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{reward}_{\mathcal{P}}$.
Else, $\mathsf{reward}_{\mathcal{P}} := 0$
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward} - \$\mathsf{smallreward} - \mathsf{reward}_{\mathcal{P}}$
Set state := ABORTED.

Timer: If state = CLAIMED and at least ΔT time elapsed since **Claim**:

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$;
Set state := ABORTED.

Else if current time $T > T_{\text{end}}$ and state $\neq$ ABORTED:

$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$.
Set state := ABORTED.

// $\mathcal{P}$ should not submit claims after $T_{end} - \Delta T$.

Figure 5: Key compromise CSC that thwarts the revoke-and-claim attack and the rushing attack.

its security such that the zero-knowledge proof ensures **simulation extractable soundness**. In brief, a one-time key generation phase is needed to generate two keys: a public evaluation key, and a public verification key. To prove a certain NP statement, an untrusted prover uses the evaluation key to compute a succinct proof; any verifier can use the public verification key to verify the proof. (The verifier in our case is the contract.) It is important that the key generation be executed confidentially, otherwise a prover can produce a valid proof for a false statement.

zk-SNARK circuits for Claim. To estimate the proof computation and verification costs required for **Claim**, we implemented the above protocol for theft of RSA-2048 and ECDSA_P256 keys, which are widely used in SSL/TLS certificates currently. The circuit has two main sub-circuits: a key-check circuit, and an encryption circuit [2] The encryption circuit was realized using RSAES-OAEP [42] with a 2048-bit key. Relying on compilers for high-level implementation of these algorithms may produce expensive circuits for the zk-SNARK proof computation. Instead, we built customized circuit generators that produce more efficient circuits. We then used the state-of-the-art zk-SNARK library [19] to obtain the evaluation results. Table 1 shows the results of the evaluation of the circuits for both single-target and multi-target contracts. The experiments were conducted on an Amazon EC2 r3.2xlarge instance with 61GB of memory and 2.5 GHz processors.

The results yield two interesting observations: i) Once a perpetrator obtains the secret key of a TLS public key, computing the zk-SNARK proof would require much less than an hour, costing less than 1 USD [5] for either single or multi-target contracts; ii) The overhead introduced by using a multi-target contract with 500 keys on the prover's side is minimal. This minimized overhead for the 500-key contract is obtained by the use of a very cheap multiplexing circuit with a secret input, while using the same components of the single-target case as is. On the other hand, in the 500-key case, the contract will have to store a larger verification key, resulting in verification times of 35msec for RSA. Further practical implementations optimizations, though, can reduce the contract verification key size and overhead.

Validation of revoked certificates. The reward function in the contract above relies on certificate revocation time, and therefore the contract needs modules that can process certificate revocation proofs, such as CRLs and OCSP responses, and verify the CA digital signatures on them. As an example, we measured the running time of `openssl verify -crl_check` command, testing the revoked certificate at [13] and the CRL last updated at [9] on May 14th, 2015, that had a size of 64KB. On average, the verification executed in about 0.011 seconds on a 2.3 GHz i7 processor. The signature algorithm was sha256WithRSAEncryption, with a 2048-bit key. Since OCSP responses can be smaller than CRLs, the verification time could be even less for OCSP.

The case of multi-target contracts. Verifying the revocation proof in the case of single-target contracts is straightforward; the contract can determine whether a revocation proof corresponds to the targeted key. In the case of multi-target contracts, however, the contract does not know which target key corresponds to the proof of key theft $\mathcal{P}$ submitted. Thus, an additional step is needed to prove that the revocation corresponds to the stolen key, and the proof must be submitted by $\mathcal{C}$. $\mathcal{C}$ can either:

2. The circuit also has other signature and encryption sub-circuits needed for the simulation extractability of the proof – see Appendix B.

- Show the plaintext (the leaked private key) corresponding to the ciphertext submitted by $\mathcal{P}$. The contract will be able to verify the correctness of the encryption and identify the revoked target key. The main drawback in this case is public leakage of the stolen private key, which may have residual value despite revocation.
- Produce a zk-SNARK proof of revocation of a public key corresponding to the private key in the ciphertext produced by $\mathcal{P}$, without revealing any more information regarding the leaked private keys.

To avoid the drawback of revealing the stolen private key, we built a zk-SNARK circuit through which $\mathcal{C}$ can prove the connection between the ciphertext submitted by the perpetrator and a target key with a secret index. For efficiency, we eliminated the need for the key-check subcircuit in **Revoke** by forcing $\mathcal{P}$ to append the secret index to the secret key before applying encryption in **Claim**. The evaluation in Table 2 illustrates the efficiency of the verification done by the contract receiving the proof, and the practicality for $\mathcal{C}$ of constructing the proof. In contrast to the case for **Claim**, the one-time key generation for this circuit must be done independently from $\mathcal{C}$, so that $\mathcal{C}$ cannot cheat the contract. We note that the **Revoke** circuit we built is invariant to the cryptosystem of the target keys.

	#threads	**RSA-2048**	**ECDSA_P256**
Key Gen.	1	394.93 sec	398.53 sec
	4	178.33 sec	162.537 sec
Eval. Key		0.74 GB	0.74 GB
Ver. Key		14.62 KB	14.62 KB
Prove[$\mathcal{C}$]	1	131.38 sec	133.88 sec
	4	**68.66 sec**	**69.036 sec**
Proof		288 B	288 B
Verification [Contract]		**0.0098 sec**	**0.0097 sec**

TABLE 2: Performance of the key-compromise zk-SNARK circuit for **Revoke** needed in the case of multi-target contract. [.] refers to the entity performing the computational work.

6. Calling-Card Crimes

As noted above, ecosystems for decentralized smart contract systems (e.g., Ethereum) are expected to incorporate services that provide authenticated data feeds, digitally signed attestations to news, facts about the physical world, etc. This powerful capability will the range of CSCs very broadly to encompass events in the physical world, as in the following example:

Example 2 (Assassination CSC). Contractor $\mathcal{C}$ posts a contract Contract for the assassination of Senator X. The contract rewards a perpetrator $\mathcal{P}$ for commission of this crime.

The contract Contract takes as input from a perpetrator $\mathcal{P}$ a commitment vcc specifying *in advance* the details (day, time, and place) of the assassination. To claim the reward, $\mathcal{P}$ decommits vcc after the assassination. To verify $\mathcal{P}$'s claim, Contract searches an authenticated data feed on news to confirm the assassination of Senator X with details matching vcc.

This example also illustrates the use of what we refer to as a *calling card*, denoted cc. A calling card is an unpredictable feature of a to-be-executed crime (e.g., in Example 2, a day, time, and place). As we show here, calling cards alongside authenticated data feeds can serve as the basis of a *general framework for a wide variety of CSCs*.

A generic construction for a CSC based on a calling card is as follows. $\mathcal{P}$ provides a commitment vcc to a calling card cc to a contract in advance. After the commission of the crime, $\mathcal{P}$ proves that cc corresponds to vcc (e.g., decommits vcc). The contract refers to some trustworthy and authenticated data feed to verify that: (1) The crime was committed and (2) The calling card cc matches the crime. If both conditions are met, the contract pays a reward to $\mathcal{P}$.

More formally, let CC be a set of possible calling cards and $\mathsf{cc} \in \mathsf{CC}$ denote a calling card. As noted above, it is anticipated that an ecosystem of authenticated data feeds will arise around smart contract systems such as Ethereum. We model a data feed as a sequence pairs emanating from a source $\mathcal{S}$, where $(s(t), \sigma(t))$ is the emission for time t. The value $s(t) \in \{0,1\}^*$ here is a piece of data (e.g., news reportage) released at time t, while $\sigma(t)$ is a corresponding digital signature; $\mathcal{S}$ has an associated private / public key pair $(\mathsf{sk}_{\mathcal{S}}, \mathsf{pk}_{\mathcal{S}})$ used to sign / verify signatures.

Note that after creation, a calling-card contract requires *no further interaction from* $\mathcal{C}$, making it hard for law enforcement to trace $\mathcal{C}$ using subsequent network traffic.

6.1. Example: website defacement contract

As an example, we specify a simple CSC SiteDeface for website defacement. The contractor $\mathcal{C}$ specifies a website url to be hacked and a statement stmt to be displayed. (For example, stmt = "Anonymous. We are Legion. We do not Forgive..." and url = whitehouse.gov.)

We assume a data feed that authenticates website content, i.e., $s(t) = (w, \mathsf{url}, t)$, where w is a representation of the webpage content and t is a timestamp, denoted for simplicity in contract time. (For efficiency, w might be a hash of and pointer to the page content.) Such an authenticated feed service might take the form of, e.g., a digitally signed version of an archive of hacked websites (e.g., zone-h.com).

We also use a special function $\mathsf{preamble}(a, b)$ that verifies $b = a||x$ for strings a, b and some x. The function SigVer does the obvious signature verification operation.

For our example, we let $\mathsf{CC} = \{0,1\}^{256}$, i.e., cc is a 256-bit string. A perpetrator $\mathcal{P}$ simply selects a calling card as $\mathsf{cc} \xleftarrow{\$} \{0,1\}^{256}$. $\mathcal{P}$ computes a commitment as $\mathsf{vcc} := \mathsf{commit}(\mathsf{cc}, \mathcal{P}; \rho)$, where commit denotes a commitment scheme, and ρ randomness. (In practice, HMAC-SHA256 is a suitable choice for easy implementation in Ethereum, as it supports SHA-256 among its core primitives.) $\mathcal{P}$ decommits by revealing all arguments to commit.

The CSC SiteDeface is shown in Figure 6.

Contract SiteDeface

Init: On receiving ($\$\mathsf{reward}$, $\mathsf{pk}_{\mathcal{S}}$, url, stmt) from some $\mathcal{C}$:

Store ($\$\mathsf{reward}$, $\mathsf{pk}_{\mathcal{S}}$, url, stmt)
Set $i := 0$, $T_{\text{start}} := T$

Commit: Upon receiving commitment vcc from some $\mathcal{P}$:

Store $\mathsf{vcc}_i := \mathsf{vcc}$ and $P_i := \mathcal{P}$; $i := i + 1$.

Claim: Upon receiving as input a tuple $(\mathsf{cc}, \rho, \sigma, w, t)$ from some $\mathcal{P}$:

Find smallest i such that $\mathsf{vcc}_i = \mathsf{commit}(\mathsf{cc}, \mathcal{P}; \rho)$.
If valid i is found:
Assert $\mathsf{stmt} \in w$
Asset $\mathsf{preamble}(\mathsf{cc}, w) = \mathtt{true}$
Assert $t \geq T_{\text{start}}$
Assert $\mathsf{SigVer}(\mathsf{pk}_{\mathcal{S}}, (w, \mathsf{url}, t), \sigma) = \mathtt{true}$
Send $\$\mathsf{reward}$ to P_i and abort.

Figure 6: CSC for website defacement

We defer formal security definitions and proofs of commission-fairness for SiteDeface, as well as issues of protocol instance composition, for the full paper version.

Remarks. SiteDeface could be implemented alternatively by having $\mathcal{P}$ generate cc as a digital signature. Our implementation, however, also accommodates short, low-entropy calling cards cc, which is important for general calling-card CSCs, as we show in the next subsection.

Implementation. Given an authenticated data feed, implementing SiteDeface would be straightforward and efficient. The main overhead lies in the **Claim** module, where the contract is supposed to compute a couple of hashes, and validate the feed signature of the retrieved website data. As illustrated earlier in Section 4, a hash function call can be computed in very short time ($4\mu sec$), while checking the signature would be more costly. For example, if the retrieved content is 100KB, the contract will need about 10msec to verify the signature, in case RSA-2048 and SHA-256 are used, which would be practical in that case.

6.2. Other calling-card crimes

Using a CSC very similar to SiteDeface, a contractor $\mathcal{C}$ can solicit many other crimes, e.g., assassination, assault, murder, sabotage, hijacking, kidnapping, denial-of-service attacks, and terrorist attacks. For successful contract construction, a perpetrator $\mathcal{P}$ must be able to designate a calling card that is reliably reported by an authenticated data feed. (If $\mathcal{C}$ is concerned about suppression of information in one source, it can of course create a CSC that references multiple sources, e.g., multiple news feeds.)

In SiteDeface, the calling card cc is high-entropy—drawn uniformly (in the ROM) from a space of size $|\mathsf{CC}| = 2^{256}$. For other crimes, the space CC can be much smaller. Suppose, for example, that cc for an assassination of a public figure X is a day and city. Then an adversary can make a sequence of online guesses at cc with corresponding commitments $\mathsf{vcc}^{(1)}, \mathsf{vcc}^{(2)}, \ldots, \mathsf{vcc}^{(n)}$ such that with high probability for relatively small n (on the order of thousands), some $\mathsf{vcc}^{(i)}$ will contain the correct value cc. (Note that commit conceals cc, but does not prevent guessing attacks against it.) These guesses, moreover, can potentially be submitted in advance of the calling call cc of a true perpetrator $\mathcal{P}$, resulting in theft of the reward and undermining commission-fairness.

There are two possible, complementary ways to address this problem. One is to enlarge the space CC by tailoring attacks to include hard-to-guess details. For example, the contract might support commitment to a one-time, esoteric pseudonym Y used to claim the attack with the media, e.g., "Police report a credible claim by a group calling itself the [Y =] 'Star-Spangled Guerilla Girls'." Or a murder might involve a rare poison (Y = Polonium-210 + strychnine).

Another option is to require a commitment vcc to carry a deposit $\$\mathsf{deposit}$ for the contract that is forfeit to $\mathcal{C}$ if there is no successful claim against vcc after a predetermined time. Treating cc as a random variable, let $p = 2^{-H_\infty[\mathsf{cc}]}$. Provided that $\$\mathsf{deposit} > p \times \$\mathsf{reward}$, adversaries are economically disincentivized from brute-force guessing of calling cards. Commission-fairness then relies on economic rationality.

7. Future Directions: Attacks and Defenses

Important research remains to be done on the scope of possible CSCs and on potential countermeasures, as we now briefly explain.

7.1. Other CSCs

The CSCs we have described are just a few examples of the broad range possible with existing technologies. Also deserving study are CSCs based on emerging or as yet not practical technologies, such as:

Password theft (using SGX): It is challenging to create a smart contract PwdTheft for theft of a password PW (or other credentials such as answers to personal questions) sufficient to access a targeted account (e.g., webmail account) A. There is no clear way for $\mathcal{P}$ to prove that PW is valid for A. Leveraging trusted hardware, however, such as Intel's pending Software Guard eXtension (SGX) set of x86-64 ISA extensions [41], it is possible to craft an incentive compatible contract PwdTheft. SGX creates a confidentiality- and integrity-protected application execution environment called an *enclave*; it protects against even a hostile OS and the owner of the computing device. SGX also supports generation of a *quote*, a digitally signed attestation to the hash of a particular executable app in an enclave and permits inclusion of app-generated text, such as an app-specific key pair $(\mathsf{sk}_{\mathsf{app}}, \mathsf{pk}_{\mathsf{app}})$. A quote proves to a remote verifier that data came from an instantiation of app on an SGX-enabled host.

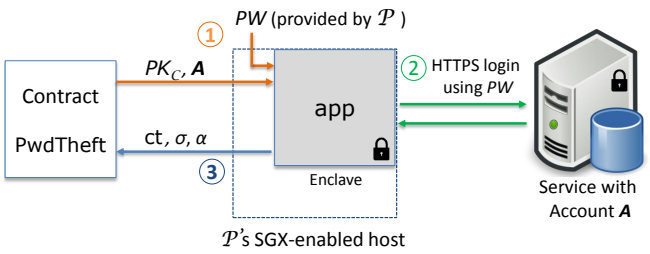

Figure 7: Diagram of execution of PwdTheft with application app running on SGX-enabled platform. The steps of operation are described in text.

We sketch the design of an executable app for PwdTheft. It does the following: (1) Ingests the password PW from $\mathcal{P}$ and ($\mathsf{pk}_{\mathcal{C}}$, A) from the contract; (2) Creates and authenticates (via HTTPS, to support source authentication) a connection to the service on which A is located; and logs into A using PW; and (3) If steps (1) and (2) are successful, sends to PwdTheft the values $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_{\mathcal{C}}}[PW]$, $\sigma = \mathsf{Sig}_{\mathsf{sk}_{\mathsf{app}}}[\mathsf{ct}]$, and a quote α for app. The functionality **Claim** in PwdTheft inputs these values and verifies σ and α, ensuring that PW is a valid password for A. At this point, PwdTheft releases a reward to $\mathcal{P}$; we omit details for this step. Figure 7 depicts the basic setup for this CSC.

After delivery of PW, $\mathcal{P}$ could cheat by changing PW, thus retaining access to A but depriving $\mathcal{C}$ of it. It is possible for app thus to include a step $(2a)$ that changes PW to a fresh, random password PW' without revealing PW' to $\mathcal{P}$. This is in effect a "proof of ignorance," a capability of trusted hardware explored in [47]. To ensure freshness, app might also ingest a timestamp, e.g., the current block header in the cryptocurrency.

Sale of 0-days: A zero-day exploit ("0-day") is a piece of code that exploits a target piece of software through a vulnerability as yet unknown to the developers and for which patches are thus unavailable. A substantial market [35] exists for the sale of 0-days as cyberweaponry [57]. Demonstrating the validity of a "0-day" without revealing it has been a persistent problem in 0-day markets, which consequently rely heavily on reputations [55].

SGX could enable proofs of validity of a 0-days: app would in this case simulate an execution environment and attest to the state of a target piece of software after execution of the 0-day. An alternative, in principle, is to construct a zk-SNARK, although, simulation of a complete execution environment would carry potentially impractical overhead.

Either technique would support the creation of a smart contract for the sale of 0-day vulnerabilities, greatly simplifying 0-day markets. Additionally, sales could be masked using an idea like that of cover claims, namely by formulating contracts *EITHER* to sell a 0-day vulnerability for $\$X$ *OR* sell $\$X$ worth of cryptocurrency. "Cover" or "decoy" contracts could then be injected into the marketplace.

7.2. CSC countermeasures

Ideas such as blacklisting "tainted" coins / transactions—those known to have been involved in criminal transactions—have been brought forward for cryptocurrencies such as Bitcoin. Such countermeasures to crime would in principle be useful to deter the creation of CSCs. Coin mixes [23], however, pose a challenge to coin tainting; if coins are mixed before they are blacklisted, then it becomes necessary also to blacklist the coins with which they were mixed, even if those coins were not involved in criminal transactions. Additionally, the notion of tainting coins has been poorly received by the cryptocurrency community because it undermines the basic cash-like property of fungibility [14], [48]. Nonetheless, this idea might deserve to be further explored if the alternative is unacceptable levels of crime.

As noted in Section 2, one approach developed as a countermeasure to crime in early (centralized) e-cash systems was a form of key escrow sometimes referred as trustee-based tracing [24], [59]. Trustee-tracing schemes permitted a single trusted party ("trustee") or a quorum of such parties to trace monetary transactions that would otherwise remaining anonymous. Two major forms of tracing were generally supported: tracing of the transactions performed by a given user and tracing of the user associated with a particular transaction. In decentralized cryptocurrencies, however, users do not register identities with an authority or set of authorities, and this very feature is one of the attraction of these systems to many users.

As an alternative directed specifically at CSCs, we propose the notion of trustee-neutralizable smart contracts. A smart contract system might be designed such that an authority or quorum of authorities is empowered to remove a contract from the blockchain. Such an approach would have a big advantage over traditional trustee-based protections, in that it *would not require users to register identities*. Whether the idea would be palatable to cryptocurrency communities and whether a broadly acceptable set of authorities could be identified are, of course, open questions.

8. Conclusion

We have demonstrated that a range of commission-fair *criminal smart contracts* (CSCs) are practical for implementation in decentralized currencies with smart contracts. We presented three, leakage of secrets, key theft, and calling-card crimes, and showed that they are efficiently implementable with existing cryptographic techniques, given suitable opcode support and the authenticated-data-feed ecosystem anticipated to arise around smart contract systems such as Ethereum. Our proposed contract PublicLeaks and its private variant can be efficiently implemented in Serpent, the Ethereum scripting language and are thus imminent possibilities. Similarly, KeyTheft would require only modest, already envisioned opcode support for SNARKs; our experiments have shown that zk-SNARKs for this contract are well within reach. Calling-card CSCs will be possible

given a sufficiently rich data-feed ecosystem. Many more CSCs are no doubt possible.

Ours is among the first academic treatment of smart contracts in distributed cryptocurrencies. It is worth emphasizing our belief that smart contracts in distributed cryptocurrencies have numerous promising, legitimate applications, such as a spectrum of new over-the-counter (OTC) financial instruments. Banning smart contracts would be neither sensible nor, in all likelihood, possible. The urgent open question raised by our work is thus how to create safeguards against the most dangerous abuses while supporting the many powerful, beneficial applications of smart contracts in distributed cryptocurrencies.

Acknowledgments

We gratefully acknowledge Rafael Pass, abhi shelat, Jonathan Katz, and Andrew Miller for helpful technical discussions regarding the cryptographic constructions and formalism. We especially thank Rafael Pass for pointing out the rushing attack. This work is funded in part by NSF grant CNS-1314857, a Sloan Fellowship, and Google Research Awards.

References

[1] http://www.smartcontract.com.
[2] http://koinify.com.
[3] https://github.com/darkwallet/darkleaks.
[4] https://en.bitcoin.it/wiki/Transaction_Malleability.
[5] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/.
[6] Augur. http://www.augur.net/.
[7] Bitcoin ransomware now spreading via spam campaigns. http://www.coindesk.com/bitcoin-ransomware-now-spreading-via-spam-campaigns/.
[8] bitoinj. https://bitcoinj.github.io/.
[9] CRL issued bby Symantec Class 3 EV SSL CA - G3. http://ss.symcb.com/sr.crl.
[10] NIST randomness beacon. https://beacon.nist.gov/home.
[11] Serpent. https://github.com/ethereum/wiki/wiki/Serpent.
[12] Skuchain. http://www.skuchain.com/.
[13] Verisign revoked certificate test page. https://test-sspev.verisign.com:2443/test-SPPEV-revoked-verisign.html. Accessed: 2015-05-15.
[14] Mt. Gox thinks it's the Fed. freezes acc based on "tainted" coins. (unlocked now). https://bitcointalk.org/index.php?topic=73385.0, 2012.
[15] Ethereum and evil. Forum post at Reddit; url: http://tinyurl.com/k8awj2j, Accessed May 2015.
[16] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure Multiparty Computations on Bitcoin. In *S & P*, 2013.
[17] J. Bates. Trojan horse: AIDS information introductory diskette version 2.0,. In E. Wilding and F. Skulason, editors, *Virus Bulletin*, pages 3–6. 1990.
[18] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *S & P*. IEEE, 2014.
[19] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, 2014.
[20] I. Bentov and R. Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.
[21] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *CCS*, 2012.
[22] V. Blue. Cryptolocker's crimewave: A trail of millions in laundered Bitcoin. *ZDNet*, 22 December 2013.
[23] J. Bonneau, A. Narayanan, A. Miller, J. Clark, J. A. Kroll, and E. W. Felten. Mixcoin: Anonymity for Bitcoin with accountable mixes. In *Financial Cryptography*, March 2014.
[24] E. F. Brickell, P. Gemmell, and D. W. Kravitz. Trustee-based tracing extensions to anonymous cash and the making of anonymous change. In *SODA*, volume 95, pages 457–466, 1995.
[25] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO '03*. 2003.
[26] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.
[27] R. Canetti. Universally composable signature, certification, and authentication. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 219–233. IEEE, 2004.
[28] R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In *CCS*, 2014.
[29] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1983.
[30] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO'92*, pages 89–105, 1993.
[31] N. Christin. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *WWW*, 2013.
[32] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, 1986.
[33] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop*, 2013.
[34] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. https://eprint.iacr.org/2015/460.
[35] S. Egelman, C. Herley, and P. C. van Oorschot. Markets for zero-day exploits: Ethics and implications. In *NSPW*. ACM, 2013.
[36] P. T. et al. Darkwallet on twitter: "DARK LEAKS coming soon. http://t.co/k4ubs16scr". Reddit: http://bit.ly/1A9UShY.
[37] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.
[38] E. F. Foundation. EFF SSL observatory. URL: https://www.eff.org/observatory, August 2010.
[39] A. Greenberg. 'Dark Wallet' is about to make Bitcoin money laundering easier than ever. http://www.wired.com/2014/04/dark-wallet/.
[40] A. Greenberg. Alleged silk road boss ross ulbricht now accused of six murders-for-hire, denied bail. *Forbes*, 21 November 2013.
[41] Intel. Intel software guard extensions programming reference. Whitepaper ref. 329298-002US, October 2014.
[42] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003. RFC 3447.
[43] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. http://eprint.iacr.org/2015/675, 2015.
[44] V. Kotov and M. Rajpal. Understanding crypto-ransomware. Bromium whitepaper, 2014.
[45] A. Krellenstein, R. Dermody, and O. Slama. Counterparty announcement. https://bitcointalk.org/index.php?topic=395761.0, January 2014.
[46] R. Kumaresan and I. Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.
[47] P. Mateus and S. Vaudenay. On tamper-resistance from a theoretical viewpoint. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 411–428. 2009.
[48] J. Matonis. Why Bitcoin fungibility is essential. *CoinDesk*, 1 Dec. 2013.
[49] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *IMC*, 2013.
[50] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *S & P*, 2013.
[51] M. Moser, R. Bohme, and D. Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCRS*, 2013.
[52] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2009.
[53] R. Pass and a. shelat. Micropayments for peer-to-peer currencies. Manuscript.
[54] K. Poulsen. Cybercrime supersite 'DarkMarket' was FBI sting, documents confirm. *Wired*, 13 Oct. 2008.

[55] J. Radianti, E. Rich, and J. Gonzalez. Using a mixed data collection strategy to uncover vulnerability black markets. In *Pre-ICIS Workshop on Information Security and Privacy*, 2007.
[56] D. Ron and A. Shamir. How did Dread Pirate Roberts acquire and protect his bitcoin wealth? In *FC*. 2014.
[57] B. Schneier. The vulnerabilities market and the future of security. *Forbes*, May 30, 2012.
[58] S. V. Solms and D. Naccache. On blind signatures and perfect crimes. *Computers Security*, 11(6):581–583, 1992.
[59] M. Stadler, J.-M. Piveteau, and J. Camenisch. Fair blind signatures. In *Eurocrypt*, pages 209–219, 1995.
[60] G. Wood. Ethereum: A secure decentralized transaction ledger. http://gavwood.com/paper.pdf, 2014.
[61] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *S & P*, 1996.

Appendix A. Cryptographic Model for Smart Contracts

To formally prove the security of our protocols, we will adopt a formal model for decentralized smart contracts, that has been recently proposed in the Hawk work [43]. We refer the readers to the Hawk manuscript for a detailed description of the model. For completeness and for this paper to be self-contained, below we restate the formal model proposed in the Hawk work [43].

We describe a formal model for smart contract execution under which we give formal proofs of security for our protocols. Our model conforms to the Universal Composability (UC) framework [26]. First, our model allows us to easily capture the temporal and pseudonym features of cryptocurrencies. In cryptocurrencies such as Bitcoin and Ethereum, time progresses in discrete intervals (known as "epochs." A smart contract program is able to query the current time, and make decisions accordingly, e.g., make a refund operation after a timeout. Second, our model captures the role of a smart contract as a party trusted for correctness but not for privacy. Third, our formalism modularizes our notations by factoring out common specifics related to the smart contract execution model, and implementing these in central wrappers.

In a real-life cryptocurrency system such as Bitcoin or Ethereum, users can make up any number of identities by generating new public keys. In our formal model, for simplicity, we assume that there can be any number of identities in the system, and that they are fixed a priori. It is easy to extend our model to capture registration of new identities dynamically. As mentioned later, we allow each identity to generate an arbitrary number of pseudonyms.

A.1. Programs, Functionalities, and Wrappers

To simply the notation for ideal functionalities and smart contracts, we introduce *wrapper* functionalities. Wrapper functionalities implement in a central place a set of common features (e.g., timer, ledger, pseudonyms) that are applicable to all ideal functionalities and contracts in our smart contract model of execution. In this way, we can modularize our notational system such that these general (and tedious) details need not be repeated across ideal functionalities and contract programs.

We introduce the following wrappers:

Ideal functionality wrapper $\mathcal{F}$: An ideal functionality $\mathcal{F}(\mathsf{idealP})$ takes in an *ideal program* denoted idealP. Specifically, the wrapper $\mathcal{F}(\cdot)$ part defines standard features such as time, pseudonyms, a public ledger, and money transfers between parties. Our ideal functionality wrapper is formally presented in Figure 8.

Contract functionality wrapper $\mathcal{G}$: A contract functionality wrapper $\mathcal{G}(\mathsf{Contract})$ takes in a *contract program* denoted $\mathsf{Contract}$, and produces a contract functionality. Our real world protocols will be defined in the $\mathcal{G}(\mathsf{Contract})$-

$\mathcal{F}(\mathsf{idealP})$ **functionality**

Given an ideal program denoted idealP, the $\mathcal{F}(\mathsf{idealP})$ functionality is defined as below:

Init: Upon initialization, perform the following:

Time. Set current time $T := 0$. Set the receive queue $\mathsf{rqueue} := \emptyset$.

Pseudonyms. Set $\mathsf{nyms} := \{(P_1, P_1), \ldots, (P_N, P_N)\}$, i.e., initially every party's true identity is recorded as a default pseudonym for the party.

Ledger. A ledger dictionary structure $\mathsf{ledger}[P]$ stores the endowed account balance for each identity $P \in \{P_1, \ldots, P_N\}$. Before any new pseudonyms are generated, only true identities have endowed account balances. Send the array $\mathsf{ledger}[]$ to the ideal adversary $\mathcal{S}$.

idealP.**Init**. Run the **Init** procedure of the idealP program.

Tick: Upon receiving `tick` from an honest party P: notify $\mathcal{S}$ of (`tick`, P). If the functionality has collected `tick` confirmations from all honest parties since the last clock tick, then

Call the **Timer** procedure of the idealP program.

Apply the adversarial permutation perm to the rqueue to reorder the messages received in the previous round.

For each $(m, \bar{P}) \in \mathsf{rqueue}$ in the permuted order, invoke the delayed actions (in gray background) defined by ideal program idealP at the activation point named *"Upon receiving message m from pseudonym $\bar{P}$"*. Notice that the program idealP speaks of pseudonyms instead of party identifiers. Set $\mathsf{rqueue} := \emptyset$.

Set $T := T + 1$

Other activations: Upon receiving a message of the form $(m, \bar{P})$ from a party P:

Assert that $(\bar{P}, P) \in \mathsf{nyms}$.

Invoke the immediate actions defined by ideal program idealP at the activation point named *"Upon receiving message m from pseudonym $\bar{P}$"*.

Queue the message by calling $\mathsf{rqueue.add}(m, \bar{P})$.

Permute: Upon receiving (`permute`, perm) from the adversary $\mathcal{S}$, record perm.

GetTime: On receiving `gettime` from a party P, notify the adversary $\mathcal{S}$ of (`gettime`, P), and return the current time T to party P.

GenNym: Upon receiving `gennym` from an honest party P: Notify the adversary $\mathcal{S}$ of `gennym`. Wait for $\mathcal{S}$ to respond with a new nym $\bar{P}$ such that $\bar{P} \notin \mathsf{nyms}$. Now, let $\mathsf{nyms} := \mathsf{nyms} \cup \{(P, \bar{P})\}$, and send $\bar{P}$ to P. Upon receiving (`gennym`, $\bar{P}$) from a corrupted party P: if $\bar{P} \notin \mathsf{nyms}$, let $\bar{P} := \mathsf{nyms} \cup \{(P, \bar{P})\}$.

Ledger operations: *// inner activation*

Transfer: Upon receiving $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r)$ from some pseudonym $\bar{P}_s$:

Notify $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r, \bar{P}_s)$ to the ideal adversary $\mathcal{S}$.

Assert that $\mathsf{ledger}[\bar{P}_s] \geq \mathsf{amount}$.
$\mathsf{ledger}[\bar{P}_s] := \mathsf{ledger}[\bar{P}_s] - \mathsf{amount}$
$\mathsf{ledger}[\bar{P}_r] := \mathsf{ledger}[\bar{P}_r] + \mathsf{amount}$

/ $\bar{P}_s, \bar{P}_r$ can be pseudonyms or true identities. Note that each party's identity is a default pseudonym for the party. */*

Expose: On receiving `exposeledger` from a party P, return ledger to the party P.

Figure 8: The $\mathcal{F}(\mathsf{idealP})$ functionality is parameterized by an ideal program denoted idealP. An ideal program idealP can specify two types of activation points, *immediate activations* and *delayed activations*. Activation points are invoked upon recipient of messages. Immediate activations are processed immediately, whereas delayed activations are collected and batch processed in the next round. The $\mathcal{F}(\cdot)$ wrapper allows the ideal adversary $\mathcal{S}$ to specify an order perm in which the messages should be processed in the next round. For each delayed activation, we use the leak notation in an ideal program idealP to define the leakage which is *immediately* exposed to the ideal adversary $\mathcal{S}$ upon recipient of the message.

$\mathcal{G}(\mathsf{Contract})$ functionality

Given a contract program denoted Contract, the $\mathcal{G}(\mathsf{Contract})$ functionality is defined as below:

Init: Upon initialization, perform the following:

A ledger data structure $\mathsf{ledger}[\bar{P}]$ stores the account balance of party $\bar{P}$. Send the entire balance ledger to $\mathcal{A}$.
Set current time $T := 0$. Set the receive queue $\mathsf{rqueue} := \emptyset$.
Run the **Init** procedure of the Contract program.
Send the Contract program's internal state to the adversary $\mathcal{A}$.

Tick: Upon receiving `tick` from an honest party, if the functionality has collected `tick` confirmations from all honest parties since the last clock tick, then

Apply the adversarial permutation perm to the rqueue to reorder the messages received in the previous round.
Call the **Timer** procedure of the Contract program.
Pass the reordered messages to the Contract program to be processed. Set $\mathsf{rqueue} := \emptyset$.
Set $T := T + 1$

Other activations:

- *Authenticated receive:* Upon receiving a message (`authenticated`, m) from party P:
  Send (m, P) to the adversary $\mathcal{A}$
  Queue the message by calling $\mathsf{rqueue.add}(m, P)$.
- *Pseudonymous receive:* Upon receiving a message of the form $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ from any party:
  Send $(m, \bar{P}, \sigma)$ to the adversary $\mathcal{A}$
  Parse $\sigma := (\mathsf{nonce}, \sigma')$, and assert $\mathsf{Verify}(\bar{P}.\mathsf{spk}, (\mathsf{nonce}, T, \bar{P}.\mathsf{epk}, m), \sigma') = 1$
  If message $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ has not been received earlier in this round, queue the message by calling $\mathsf{rqueue.add}(m, \bar{P})$.
- *Anonymous receive:* Upon receiving a message (`anonymous`, m) from party P:
  Send m to the adversary $\mathcal{A}$
  If m has not been seen before in this round, queue the message by calling $\mathsf{rqueue.add}(m)$.

Permute: Upon receiving (`permute`, perm) from the adversary $\mathcal{A}$, record perm.
Expose: On receiving `exposestate` from a party P, return the functionality's internal state to the party P. Note that this also implies that a party can query the functionality for the current time T.

Ledger operations: *// inner activation*

Transfer: Upon recipient of $(\texttt{transfer}, \mathsf{amount}, \bar{P}_r)$ from some pseudonym $\bar{P}_s$:

Assert $\mathsf{ledger}[\bar{P}_s] \geq \mathsf{amount}$
$\mathsf{ledger}[\bar{P}_s] := \mathsf{ledger}[\bar{P}_s] - \mathsf{amount}$
$\mathsf{ledger}[\bar{P}_r] := \mathsf{ledger}[\bar{P}_r] + \mathsf{amount}$

Figure 9: The $\mathcal{G}(\mathsf{Contract})$ functionality is parameterized by a contract program denoted Contract. The $\mathcal{G}(\cdot)$ wrapper mainly performs the following: *i)* exposes all of its internal states and messages received to the adversary; *ii)* makes the functionality time-aware: messages received in one round and queued and processed in the next round. The $\mathcal{G}(\cdot)$ wrapper allows the adversary to specify an ordering to the messages received by the contract in one round.

hybrid world. Our contract functionality wrapper is formally presented in Figure 9.

Like the ideal functionality wrapper $\mathcal{F}(\cdot)$, the contract wrapper also implements standard features such as a timer, a global ledger, and money transfers between parties. In addition, the contract wrapper also models the specifics of the smart contract execution model. We point out the following important facts about the $\mathcal{G}(\cdot)$ wrapper:

- *Trusted for correctness but not privacy.* The contract functionality wrapper $\mathcal{G}(\cdot)$ stipulates that a smart contract is trusted for correctness but not for privacy. In particular, the contract wrapper exposes the contract's internal state to any party that makes a query.
- *Time and batched processing of messages.* In popular decentralized cryptocurrencies such as Bitcoin and Ethereum, time progresses in discrete intervals marked by the creation of each new block. Intuitively, our $\mathcal{G}(\cdot)$ wrapper captures the following fact. In each round (i.e., block interval), the smart contract program may receive multiple messages (also referred to as transactions in the cryptocurrency literature). The order of processing these transactions is determined by the miner who mines the next block. In our model, we allow the adversary to specify an ordering of the messages collected in a round, and our contract program will then process the messages in this adversary-specified ordering.
- *Rushing adversary.* The contract wrapper $\mathcal{G}(\cdot)$ naturally models a rushing adversary. Specifically, the adversary

can first see all messages sent to the contract by honest parties, and then decide its own messages for this round, as well as an ordering in which the contract should process the messages in the next round. Modeling a rushing adversary is important, since it captures a class of well-known front-running attacks, e.g., those that exploit transaction malleability [4], [18]. For example, in a "rock, paper, scissors" game, if inputs are sent in the clear, an adversary can decide its input based on the other party's input. An adversary can also try to maul transactions submitted by honest parties to potentially redirect payments to itself. Since our model captures a rushing adversary, we can write ideal functionalities that preclude such front-running attacks.

Protocol wrapper Π: Our protocol wrapper allows us to modularize the presentation of user protocols. Our protocol wrapper is formally presented in Figure 10.

Simulator wrapper $\mathcal{S}$: The ideal adversary $\mathcal{S}$ can typically be obtained by applying the simulator wrapper $\mathcal{S}(\cdot)$ to the user-defined portion of the simulator simP. The simulator wrapper modularizes the simulator construction by factoring out the common part of the simulation pertaining to all protocols in this model of execution.

The simulator is defined formally in Figure 11.

A.2. Modeling Time

At a high level, we express time in a way that conforms to the Universal Composability framework [26]. In the ideal world execution, time is explicitly encoded by a variable T in an ideal functionality $\mathcal{F}(\mathsf{idealP})$. In the real world execution, time is explicitly encoded by a variable T in contract functionality $\mathcal{G}(\mathsf{Contract})$. Time progresses in rounds. The environment $\mathcal{E}$ choses when to advance the timer.

We assume the following convention: to advance the timer, the environment $\mathcal{E}$ sends a "tick" message to all honest parties. Honest parties' protocols would then forward this message to $\mathcal{F}(\mathsf{idealP})$ in the ideal-world execution, or to the $\mathcal{G}(\mathsf{Contract})$ functionality in the real-world execution. On collecting "tick" messages from all honeset parties, the $\mathcal{F}(\mathsf{idealP})$ or $\mathcal{G}(\mathsf{Contract})$ functionality would then advance the time $T := T + 1$. The functionality also allows parties to query the current time T.

A.3. Modeling Pseudonyms

We model a notion of "pseudonymity" that provides a form of privacy, similar to that provided by typical cryptocurrencies such as Bitcoin. Any user can generate an arbitrary (polynomially-bounded) number of pseudonyms, and each pseudonym is "owned" by the party who generated it. The correspondence of pseudonyms to real identities is hidden from the adversary.

Effectively, a pseudonym is a public key for a digital signature scheme; the corresponding private key is known by the party who "owns" the pseudonym. The public contract functionality allows parties to publish authenticated messages that are bound to a pseudonym of their choice. Thus each interaction with the public contract is, in general, associated with a pseudonym, but not to a user's real identity.

We abstract away the details of pseudonym management by implementing them in our wrappers. This allows user-defined applications to be written very simply, as though using ordinary identities, while enjoying the privacy benefits of pseudonymity.

Our wrapper provides a user-defined hook, "gennym", that is invoked each time a party creates a pseudonym. This allows the application to define an additional per-pseudonym payload, such as application-specific public keys. From the point of view of the application, this is simply an initialization subroutine invoked once for each participant.

Our wrapper provides several means for users to communicate with a contract. The most common way is for a user to publish an authenticated message associated with one of their pseudonyms, as described above. Additionally, "anonsend" allows a user to publish a message without reference to any pseudonym at all.

In spite of pseudonymity, it is sometimes desirable to assign a particular user to a specific role in a contract (e.g., "auction manager"). The alternative is to assign roles on a "first-come first-served" basis (e.g., as the bidders in an auction). To this end, we allow each party to define generate a single "default" pseudonym which is publicly-bound to their real identity. We allow applications to make use of this through a convenient abuse of notation, by simply using a party identifier as a parameter or hardcoded string. Strictly speaking, the pseudonym string is not determined until the "gennym" subroutine is executed; the formal interpretation is that whenever such an identity is used, the default pseudonym associated with the identity is fetched from the contract. (This approach is effectively the same as taken by Canetti [27], where a functionality $\mathcal{F}_{\mathtt{CA}}$ allows each party to bind their real identity to a single public key of their choice).

A.4. Modeling Money

We model money as a public ledger, which associates quantities of money to pseudonyms. Users can transfer funds to each other (or among their own pseudonyms) by sending "transfer" messages to the public contract (as with other messages, these are delayed util the next round and may be delivered in any order). The ledger state is public knowledge, and can be queried immediately using the "exposeledger" instruction.

There are many conceivable policies for introducing new currency into such a system: for example, Bitcoin "mints" new currency as a reward for each miner who solves a proof-of-work puzzles. We take a simple approach of defining an arbitrary, publicly visible (i.e., common knowledge) initial allocation that associates a quantity of money to each party's real identity. Except for this initial allocation, no money is created or destroyed.

$\Pi(\mathsf{prot})$ **protocol wrapper in the** $\mathcal{G}(\mathsf{Contract})$**-hybrid world**

Given a party's local program denoted prot, the $\Pi(\mathsf{prot})$ functionality is defined as below:

Pseudonym related:

GenNym: Upon receiving input `gennym` from the environment $\mathcal{E}$, generate $(\mathsf{epk}, \mathsf{esk}) \leftarrow \mathsf{Keygen}_{\mathrm{enc}}(1^\lambda)$, and $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{Keygen}_{\mathrm{sign}}(1^\lambda)$. Call $\mathsf{payload} := \mathsf{prot}.\mathbf{GenNym}(1^\lambda, (\mathsf{epk}, \mathsf{spk}))$. Store $\mathsf{nyms} := \mathsf{nyms} \cup \{(\mathsf{epk}, \mathsf{spk}, \mathsf{payload})\}$, and output $(\mathsf{epk}, \mathsf{spk}, \mathsf{payload})$ as a new pseudonym.

Send: Upon receiving internal call (`send`, m, $\bar{P}$):

If $\bar{P} == P$: send (`authenticated`, m) to $\mathcal{G}(\mathsf{Contract})$. *// this is an authenticated send*

Else, *// this is a pseudonymous send*

Assert that pseudonym $\bar{P}$ has been recorded in nyms;

Query current time T from $\mathcal{G}(\mathsf{Contract})$. Compute $\sigma' := \mathsf{Sign}(\mathsf{ssk}, (\mathsf{nonce}, T, \mathsf{epk}, m))$ where ssk is the recorded secret signing key corresponding to $\bar{P}$, nonce is a freshly generated random string, and epk is the recorded public encryption key corresponding to $\bar{P}$. Let $\sigma := (\mathsf{nonce}, \sigma')$.

Send $(\texttt{pseudonymous}, m, \bar{P}, \sigma)$ to $\mathcal{G}(\mathsf{Contract})$.

AnonSend: Upon receiving internal call (`anonsend`, m, $\bar{P}$): send (`anonymous`, m) to $\mathcal{G}(\mathsf{Contract})$.

Timer and ledger transfers:

Transfer: Upon receiving input (`transfer`, \$amount, $\bar{P}_r$, $\bar{P}$) from the environment $\mathcal{E}$:

Assert that $\bar{P}$ is a previously generated pseudonym.

Send $\left(\texttt{transfer}, \$\mathsf{amount}, \bar{P}_r\right)$ to $\mathcal{G}(\mathsf{C})$ as pseudonym $\bar{P}$.

Tick: Upon receiving `tick` from the environment $\mathcal{E}$, forward the message to $\mathcal{G}(\mathsf{Contract})$.

Other activations:

Act as pseudonym: Upon receiving any input of the form $(m, \bar{P})$ from the environment $\mathcal{E}$:

Assert that $\bar{P}$ was a previously generated pseudonym.

Pass $(m, \bar{P})$ the party's local program to process.

Others: Upon receiving any other input from the environment $\mathcal{E}$, or any other message from a party: Pass the input/message to the party's local program to process.

Figure 10: Protocol wrapper.

A.5. Conventions for Writing Programs

Thanks to our wrapper-based modularized notational system, The ideal program and the contract program are the main locations where user-supplied, custom program logic is defined. We use the following conventions for writing the ideal program and the contract program.

Timer activation points. Every time $\mathcal{F}(\mathsf{idealP})$ or $\mathcal{G}(\mathsf{Contract})$ advances the timer, it will invoke a **Timer** interrupt call. Therefore, by convention, we allow the ideal program or the contract program can define a **Timer** activation point. Timeout operations (e.g., refunding money after a certain timeout) can be implemented under the **Timer** activation point.

Delayed processing in ideal programs. When writing the contract program, every message received by the contract program is already delayed by a round due to the $\mathcal{G}(\cdot)$ wrapper.

When writing the ideal program, we introduce a simple convention to denote delayed computation. Program instructions that are written in gray background denote computation that does not take place immediately, but is deferred to the beginning of the next timer click. This is a convenient shorthand because in our real-world protocol, effectively any computation done by a contract functionality will be delayed. Formally, delayed processing can be implemented simply by storing state and invoking the delayed program instructions on the next **Timer** click. To avoid ambiguity, we assume that by convention, the delayed instructions are invoked at the beginning of the **Timer** call. In other words, upon the next timer click, the delayed instructions are executed first.

Pseudonymity. All party identifiers that appear in ideal programs, contract programs, and user-side programs by default refer to *pseudonyms*. When we write "upon receiving message from *some* P", this accepts a message from any pseudonym. Whenever we write "upon receiving message from P", without the keyword *some*, this accepts a message from a fixed pseudonym P, and typically which pseudonym we refer to is clear from the context.

Whenever we write "send m to $\mathcal{G}(\mathsf{Contract})$ as nym P" inside a user program, this sends an internal message ("send", m, P) to the protocol wrapper Π. The protocol wrapper will then authenticate the message appropriately under pseudonym P. When the context is clear, we avoid writing "as nym P", and simply write "send m to $\mathcal{G}(\mathsf{Contract})$". Our formal system also allows users to send messages anonymously to a contract – although this option

$\mathcal{S}(\mathsf{simP})$

Init. The simulator $\mathcal{S}$ simulates a $\mathcal{G}(\mathsf{contract})$ instance internally. Here $\mathcal{S}$ calls $\mathcal{G}(\mathsf{contract})$.**Init** to initialize the internal states of the contract functionality. $\mathcal{S}$ also calls simP.**Init**.

Simulating honest parties.

- **Tick:** Environment $\mathcal{E}$ sends input `tick` to an honest party P: simulator $\mathcal{S}$ receives notification (`tick`, P) from the ideal functionality. Simulator forwards the tick message to the simulated $\mathcal{G}(\mathsf{contract})$ functionality.
- **GenNym:** Environment $\mathcal{E}$ sends input `gennym` to an honest party P: simulator $\mathcal{S}$ receives notification `gennym` from the ideal functionality. Simulator $\mathcal{S}$ honestly generates an encryption key and a signing key as defined in Figure 10, and remembers the corresponding secret keys. Simulator $\mathcal{S}$ now calls $\mathsf{simP}.\mathbf{GenNym}(\mathsf{epk}, \mathsf{spk})$ and waits for the returned value $\mathsf{payload}$. Finally, the simulator passes the nym $\bar{P} = (\mathsf{epk}, \mathsf{spk}, \mathsf{payload})$ to the ideal functionality.
- **Other activations.** // *From the inner* idealP
  If ideal functionality sends (`transfer`, $\$\mathsf{amount}$, P_r, P_s), then update the ledger in the simulated $\mathcal{G}(\mathsf{Contract})$ instance accordingly.
  Else, forward the message to the inner simP.

Simulating corrupted parties.

- **Permute:** Upon receiving (`permute`, perm) from the environment $\mathcal{E}$, forward it to the internally simulated $\mathcal{G}(\mathsf{contract})$ and the ideal functionality.
- **Expose.** Upon receiving `exposestate` from the environment $\mathcal{E}$, expose all states of the internally simulated $\mathcal{G}(\mathsf{contract})$.
- **Other activations.**
  - Upon receiving (`authenticated`, m) from the environment $\mathcal{E}$ on behalf of corrupted party P: Forward to internally simulated $\mathcal{G}(\mathsf{contract})$. If the message is of the format (`transfer`, $\$\mathsf{amount}$, P_r, P_s), then forward it to the ideal functionality. Otherwise, forward to simP.
  - Upon receiving (`pseudonymous`, m, $\bar{P}$, σ) from the environment $\mathcal{E}$ on behalf of corrupted party P: Forward to internally simulated $\mathcal{G}(\mathsf{contract})$. Now, assert that σ verifies just like in $\mathcal{G}(\mathsf{contract})$. If the message is of the format (`transfer`, $\$\mathsf{amount}$, P_r, P_s), then forward it to the ideal functionality. Else, forward to simP.
  - Upon receiving (`anonymous`, m) from the environment $\mathcal{E}$ on behalf of corrupted party P: Forward to internally simulated $\mathcal{G}(\mathsf{contract})$. If the message is of the format (`transfer`, $\$\mathsf{amount}$, P_r, P_s), then forward it to the ideal functionality. Else, forward to simP.

Figure 11: Simulator wrapper.

will not be used in this paper.

Ledger and money transfers. A public dedger is denoted ledger in our ideal programs and contract programs. When a party sends $\$\mathsf{amt}$ to an ideal program or a contract program, this represents an ordinary message transmission. Money transfers only take place when ideal programs or contract programs update the public ledger ledger. For clarity, a variable beginning with the \$ sign means that this variable denotes some amount of money. Otherwise, there is no special mathematical significance related to the \$ sign.

Appendix B. Preliminaries

Notation. In the remainder of the paper, $f(\lambda) \approx g(\lambda)$ means that there exists a negligible function $\nu(\lambda)$ such that $|f(\lambda) - g(\lambda)| < \nu(\lambda)$.

B.1. Non-Interactive Zero-Knowlege Proofs

A non-interactive zero-knowlege proof system (NIZK) for an NP language $\mathcal{L}$ consists of the following algorithms:

- $\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L})$, also written as $\mathsf{crs} \leftarrow \mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda, \mathcal{L})$: Takes in a security parameter λ, a description of the language $\mathcal{L}$, and generates a common reference string crs.
- $\pi \leftarrow \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w)$: Takes in crs, a statement stmt, a witness w such that $(\mathsf{stmt}, w) \in \mathcal{L}$, and produces a proof π.
- $b \leftarrow \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi)$: Takes in a crs, a statement stmt, and a proof π, and outputs 0 or 1, denoting accept or reject.

Perfect completeness. A NIZK system is said to be perfectly complete, if an honest prover with a valid witness can always convince an honest verifier. More formally, for any $(\mathsf{stmt}, w) \in R$, we have that

$$\Pr\left[\begin{array}{c} \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}),\ \pi \leftarrow \mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w) : \\ \mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1 \end{array}\right] = 1$$

Computational zero-knowlege. Informally, a NIZK system is computationally zero-knowledge, if the proof does not reveal any information about the witness to any polynomial-time adversary. More formally, a NIZK system is said to computationally zero-knowledge, if there exists

a polynomial-time simulator $S = (\widehat{\mathcal{K}}, \widehat{\mathcal{P}})$, such that for all non-uniform polynomial-time adversary $\mathcal{A}$, we have that

$$\begin{aligned} & \Pr\left[\mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\mathcal{P}(\mathsf{crs},\cdot,\cdot)} = 1\right] \\ \approx \quad & \Pr\left[(\widehat{\mathsf{crs}}, \tau) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L}) : \mathcal{A}^{\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}},\tau,\cdot,\cdot)} = 1\right] \end{aligned}$$

In the above, $\widehat{\mathcal{P}}_1(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt}, w)$ verifies that $(\mathsf{stmt}, w) \in \mathcal{L}$, and if so, outputs $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$ which simulates a proof without knowing a witness.

Computational soundness. A NIZK scheme for the language $\mathcal{L}$ is said to be computationally sound, if for all polynomial-time adversaries $\mathcal{A}$,

$$\Pr\left[\begin{array}{l} \mathsf{crs} \leftarrow \mathcal{K}(1^\lambda, \mathcal{L}), (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}(\mathsf{crs}) : \\ (\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi) = 1) \wedge (\mathsf{stmt} \notin \mathcal{L}) \end{array}\right] \approx 0$$

Simulation sound extractability. Simulation sound extractability says that even after seeing many simulated proofs, whenever the adversary makes a new proof, a simulator is able to extract a witness. Simulation extractability implies simulation soundness and non-malleability, since if the simulator can extract a valid witness from an adversary's proof, the statement must belong to the language. More formally, a NIZK system is said to be simulation sound extractable, if there exist polynomial-time algorithms $(\widehat{\mathcal{K}}, \widehat{\mathcal{P}}, \mathcal{E})$, such that for any polynomial-time adversary $\mathcal{A}$, it holds that

$$\Pr\left[\begin{array}{l} (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L}); \\ (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}},\tau,\cdot)}(\widehat{\mathsf{crs}}, \mathsf{ek}); \\ w \leftarrow \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, \mathsf{stmt}, \pi) : \mathsf{stmt} \notin Q \text{ and} \\ \qquad (\mathsf{stmt}, w) \notin \mathcal{L} \text{ and } V(\widehat{\mathsf{crs}}, \mathsf{stmt}, \pi) = 1 \end{array}\right] \approx 0$$

In the above, Q is the list of simulation queries. Here the $\widehat{\mathcal{K}}$ is identical to the zero-knowledge simulation setup algorithm when restricted to the first two terms.

Note that in the above definition, the adversary may be able to fake a (different) proof for a statement that has been queried, however, it is not able to forge a proof for any other invalid statement. There is a natural strengthening of the above notion where the adversary cannot even fake a different proof for a statement queried. In our paper, however, the weaker notion defined above would suffice.

B.2. Simulation Sound Extractable Proofs from Ordinary NIZKs

We give a generic transformation that turns an ordinary NIZK into one that satisfies simulation sound extractability. In our implementation, we use Zero-Knowledge Succinct Non-interactive ARguments of Knowledge (SNARKs) as the underlying NIZK, and apply our transformation here to make it a NIZK with simulation sound extractability. For this reason, in the transformation described below, we simply use snark to denote the underlying NIZK.

- $\mathcal{K}(1^\lambda, \mathcal{L})$: Run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \Sigma.\mathsf{Gen}(1^\lambda)$. Run $(\mathsf{pk}_e, \mathsf{sk}_e) \leftarrow \mathsf{KeyGen}_{\mathsf{enc}}(1^\lambda)$.
  Let $\mathcal{L}'$ be the following language: $((\mathsf{stmt}, c), (r, w, \sigma)) \in \mathcal{L}'$ *iff*
  $$(c = \mathsf{Enc}(\mathsf{pk}_e, (w, \sigma), r)) \wedge ((\mathsf{stmt}, w) \in \mathcal{L} \vee (\Sigma.\mathcal{V}(\mathsf{pk}, \mathsf{stmt}, \sigma) = 1))$$
  Run $\mathsf{snark.crs} \leftarrow \mathsf{snark}.\mathcal{K}(1^\lambda, \mathcal{L}')$.
  Publish $\mathsf{crs} := (\mathsf{snark.crs}, \mathsf{pk}, \mathsf{pk}_e)$ as the common reference string.
- $\mathcal{P}(\mathsf{crs}, \mathsf{stmt}, w)$: Parse $\mathsf{crs} := (\mathsf{snark.crs}, \mathsf{pk})$. Choose random r, and compute $c := \mathsf{Enc}(\mathsf{pk}_e, (w, \sigma), r)$.
  Call $\pi := \mathsf{snark}.\mathcal{P}(\mathsf{snark.crs}, (\mathsf{stmt}, c), (r, w, \perp))$, and output $\pi' := (c, \pi)$.
- $\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi')$: Parse $\pi' := (c, \pi)$, and output $\mathsf{snark}.\mathcal{V}(\mathsf{snark.crs}, (\mathsf{stmt}, c), \pi)$.

Theorem 1. Assume that the SNARK scheme satisfies perfect completeness, computational soundness, and computational zero-knowlege, and that the encryption scheme is perfectly correct, then the above construction is a zero-knowledge proof system satisfying perfect completeness, computational zero-knowledge, and simulation sound extractability.

Proof: The proofs of perfect completeness and computational zero-knowledge are obvious. We now show that this transformation gives a simulation sound extractable NIZK. We construct the following simulation and extractor:

- $\widehat{\mathcal{K}}(1^\lambda, \mathcal{L})$: Run the honest $\mathcal{K}$ algorithm, but retain the signing key sk as the simulation trapdoor $\tau := \mathsf{sk}$. The extraction key $\mathsf{ek} := \mathsf{sk}_e$, the simulated $\widehat{\mathsf{crs}} := \mathsf{crs} = (\mathsf{snark.crs}, \mathsf{pk}, \mathsf{pk}_e)$.
- $\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$: the simulator calls
  $$\pi := \mathsf{snark}.\mathcal{P}(\mathsf{snark.crs}, (\mathsf{stmt}, c), (\perp, \perp, \sigma))$$
  where c is an encryption of 0, and
  $$\sigma := \Sigma.\mathsf{Sign}(\mathsf{sk}, \mathsf{stmt})$$
  Output (c, π).
- $\mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, \mathsf{stmt}, \pi')$: parse $\pi' := (c, \pi)$, and let $(w, \sigma) := \mathsf{Dec}(\mathsf{sk}_e, c)$. Output w.

We now show that no polynomial-time adversary $\mathcal{A}$ can win the simulation sound extractable game except with negligible probability. Given that the encryption scheme is perfectly correct, and that snark is computationally sound, the witness (w, σ) decrypted by $\mathcal{E}$ must satisfy one of the following two cases except with negligible probability: 1) w is a valid witness for stmt under language $\mathcal{L}$; or 2) σ is a valid signature for stmt. If stmt has not been queried by the adversary $\mathcal{A}$, then it must be that w is a valid witness for stmt, since otherwise, the simulator can easily leverage the adversary to break the security of the signature scheme.

Appendix C. Formal Protocols for Key Theft Contract

C.1. Ideal Program for the Naive Key Theft

The ideal program for the naive key theft contract is given in Figure 12. We stress that here, this naive key theft

Ideal-NaiveKeyTheft

Init: Set state := INIT.

Create: Upon recipient of ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}$, T_{end}) from some contractor $\mathcal{C}$:

Notify ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}$, T_{end}, $\mathcal{C}$) to $\mathcal{S}$.

Assert ledger[$\mathcal{C}$] $\geq$ \$reward.
ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] − \$reward
state := CREATED.

Intent: On recv ("intent", $\mathsf{sk}_{\mathcal{V}}$) from some perpetrator $\mathcal{P}$:

Assert state = CREATED.
Notify ("intent", $\mathcal{P}$) to $\mathcal{S}$.
Assert this is the first "intent" received from $\mathcal{P}$.

Store $(\mathcal{P}, \mathsf{sk}_{\mathcal{V}})$.

Claim: Upon recipient of ("claim") from $\mathcal{P}$:

Assert state = CREATED.
Assert that $\mathcal{P}$ has sent ("intent", $\mathsf{sk}_{\mathcal{V}}$) earlier.
Assert match($\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}$) = 1
Notify ("claim", $\mathcal{P}$) to $\mathcal{S}$.
If $\mathcal{C}$ is corrupted, send $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{S}$.

ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + \$reward
Send $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{C}$
Set state := CLAIMED.
/ reward goes to 1st successful claim*/*

Timer: If state = CREATED and current time $T > T_{\text{end}}$:

Set ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] + \$reward
Set state := ABORTED.

Figure 12: Ideal program for naive key theft. This version of the ideal program defends against the rushing attack, but does not protect against the revoke-and-claim attack.

ideal program is different from the strawman example in the main body (Figure 4). For ease of understanding, Figure 4 in the main body is prone to a rushing attack by a corrupted contractor. Here, our naive key theft ideal program secures against the rushing attack – however, this naive key theft ideal program is still prone to the revoke-and-claim attack (see Section 5.1). We will fix the revoke-and-claim attack later in Appendix C.4

Remarks. We make the following remarks about this ideal functionality:

- All bank balances are visible to the public.
- Bank transfers are guaranteed to be correct.
- The ideal functionality captures transaction non-malleability, and precludes any front-running attack, since our real-world execution model assumes a rushing adversary.

C.2. Full Protocol for Naive Key Theft

The contract and full protocols for naive key theft are given in Figures 13 and 14. Specifically, Figure 13 is a repeat of Figure 4 for the readers' convenience.

Theorem 2. Assume that the encryption scheme (Enc, Dec) is perfectly correct and semantically secure, the NIZK scheme is perfectly complete, computationally zero-knowledge and simulation sound extractable, the commitment scheme comm is adaptively secure, then the above protocol securely emulates $\mathcal{F}$(Ideal-NaiveKeyTheft).

Contract-NaiveKeyTheft

Init: Set state := INIT. Let crs := $\mathsf{KeyGen}_{\text{nizk}}(1^{\lambda})$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Upon receiving ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}$, T_{end}) from some contractor $\mathcal{C} := (\mathsf{pk}_{\mathcal{C}}, \ldots)$:

Assert state = INIT.
Assert ledger[$\mathcal{C}$] $\geq$ \$reward.
ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] − \$reward.
Set state := CREATED.

Intent: Upon receiving ("intent", cm) from some purported perpetrator $\mathcal{P}$:

Assert state = CREATED.
Assert that $\mathcal{P}$ did not send "intent" earlier.
Store cm, $\mathcal{P}$.

Claim: Upon receiving ("claim", ct, π, s) from $\mathcal{P}$:

Assert state = CREATED.
Assert $\mathcal{P}$ sent ("intent", cm) earlier such that cm := comm(ct$||\pi, s$).
Assert that π is a valid NIZK proof (under crs) for the following statement:

$$\exists r, \mathsf{sk}_{\mathcal{V}} \text{ s.t. } \mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_{\mathcal{C}}, (\mathsf{sk}_{\mathcal{V}}, \mathcal{P}), r) \text{ and } \mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}) = \texttt{true}$$

ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + \$reward.
Send ("claim", ct) to the contractor $\mathcal{C}$.
Set state := CLAIMED.

Timer: If state = CREATED and current time $T > T_{\text{end}}$:

ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] + \$reward
state := ABORTED

Figure 13: A naïve, flawed key theft contract (lacking incentive compatibility). The notation $\mathsf{pk}_{\mathcal{C}}$ serves as a shorthand for $\mathcal{C}$.epk. This figure is a repeat of Figure 4 for the readers' convenience.

C.3. Proofs for Naive Key Theft Contract

We now prove Theorem 2. For any real-world adversary $\mathcal{A}$, we construct an ideal-world simulator $\mathcal{S}$, such that no polynomial-time environment $\mathcal{E}$ can distinguish whether it is in the real or ideal world. We first describe the construction of the simulator $\mathcal{S}$ and then argue the indistinguishability of the real and ideal worlds.

C.3.1. Ideal-World Simulator. Due to Canetti [26], it suffices to construct a simulator $\mathcal{S}$ for the dummy adversary that simply passes messages to and from the environment $\mathcal{E}$. The ideal-world simulator $\mathcal{S}$ also interacts with the

Prot-NaiveKeyTheft

Contractor $\mathcal{C}$:

Create: Upon receiving input ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}$, $\mathcal{C}$):

Send ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}$) to $\mathcal{G}$(Contract-NaiveKeyTheft).

Claim: Upon receiving a message ("claim", ct) from $\mathcal{G}$(Contract-NaiveKeyTheft):

Decrypt and output $m := \mathsf{Dec}(\mathsf{sk}_{\mathcal{C}}, \mathsf{ct})$.

Perpetrator $\mathcal{P}$:

Intent: Upon receiving input ("intent", $\mathsf{sk}_{\mathcal{V}}$, $\mathcal{P}$):

Assert $\mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}) = \mathsf{true}$

Compute $\mathsf{ct} := \mathsf{Enc}(\mathsf{pk}_{\mathcal{C}}, (\mathsf{sk}_{\mathcal{V}}, \mathcal{P}), s)$ where s is randomly chosen.

Compute a NIZK proof π for the following statement:

$$\exists r, \mathsf{sk}_{\mathcal{V}} \text{ s.t. } \mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_{\mathcal{C}}, (\mathsf{sk}_{\mathcal{V}}, \mathcal{P}), r) \text{ and } \mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}) = \mathtt{true}$$

Let $\mathsf{cm} := \mathsf{comm}(\mathsf{ct}||\pi, s)$ for some random $s \in \{0,1\}^{\lambda}$.

Send ("intent", cm) to $\mathcal{G}$(Contract-NaiveKeyTheft).

Claim: Upon receiving input ("claim"):

Assert an "intent" message was sent earlier.

Send ("claim", ct, π, s) to $\mathcal{G}$(Contract-NaiveKeyTheft).

Figure 14: User-side programs for naive key theft. The notation $\mathsf{pk}_{\mathcal{C}}$ serves as a short-hand for $\mathcal{C}$.epk.

$\mathcal{F}$(Ideal-NaiveKeyTheft) ideal functionality. Below we construct the user-defined portion of our simulator simP. Our ideal adversary $\mathcal{S}$ can be obtained by applying the simulator wrapper $\mathcal{S}$(simP). The simulator wrapper modularizes the simulator construction by factoring out the common part of the simulation pertaining to all protocols in this model of execution.

Init. The simulator simP runs $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \mathsf{NIZK}.\widehat{\mathcal{K}}(1^{\lambda})$, and gives $\widehat{\mathsf{crs}}$ to the environment $\mathcal{E}$, and retains the trapdoor τ.

Simulating honest parties. When the environment $\mathcal{E}$ sends inputs to honest parties, the simulator $\mathcal{S}$ needs to simulate messages that corrupted parties receive, from honest parties or from functionalities in the real world. The honest parties will be simulated as below.

- Environment $\mathcal{E}$ sends input ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}, \mathcal{C}$) to an honest contractor $\mathcal{C}$: Simulator simP receives ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}, \mathcal{C}$) from $\mathcal{F}$(Ideal-NaiveKeyTheft). simP forwards the message to the simulated inner contract functionality $\mathcal{G}$(Contract-NaiveKeyTheft), as well as to the environment $\mathcal{E}$.
- Environment $\mathcal{E}$ sends input ("intent", $\mathsf{sk}_{\mathcal{V}}$) to an honest perpetrator $\mathcal{P}$: Simulator simP receives notification from the ideal functionality $\mathcal{F}$(Ideal-NaiveKeyTheft) without seeing $\mathsf{sk}_{\mathcal{V}}$. Simulator simP now computes ct to be an encryption of the 0 vector. simP then simulates the NIZK π. simP now computes the commitment cm honestly. simP sends ("intent",cm) to the simulated $\mathcal{G}$(Contract-NaiveKeyTheft) functionality, and simulates the contract functionality in the obvious manner.
- Environment $\mathcal{E}$ sends input ("claim") to an honest perpetrator $\mathcal{P}$:

Case 1: Contractor $\mathcal{C}$ is honest. simP sends the ("claim", ct, π, r) values to the internally simulated $\mathcal{G}$(Contract-NaiveKeyTheft) functionality, where ct and π are the previously simulated values and r is the randomness used in the commitment cm earlier.

Case 2: Contractor $\mathcal{C}$ is corrupted. simP receives $\mathsf{sk}_{\mathcal{V}}$ from $\mathcal{F}$(Ideal-NaiveKeyTheft).
simP computes (ct', π') terms using the honest algorithm. simP now explains the commitment cm to the correctly formed (ct', π') values. Notice here we rely the commitment scheme being adaptively secure. Suppose the corresponding randomness is r' simP now sends ("claim", ct', π', r') to the internally simulated $\mathcal{G}$(Contract-NaiveKeyTheft) functionality, and simulates the contract functionality in the obvious manner.

Simulating corrupted parties. The following messages are sent by the environment $\mathcal{E}$ to the simulator $\mathcal{S}$(simP) which then forwards it onto simP. All of the following messages received by simP are of the "pseudonymous" type, we therefore omit writing "pseudonymous".

- simP receives an intent message ("intent", cm): forward it to the internally simulated $\mathcal{G}$(Contract-NaiveKeyTheft) functionality,
- simP receives a claim message ("claim", $\mathsf{ct}, \pi, r, \mathcal{P}$): If π verifies, simulator simP runs the NIZK's extraction algorithm, and extracts a set of witnesses including $\mathsf{sk}_{\mathcal{V}}$. $\mathcal{S}$ now sends ("claim", $\mathsf{sk}_{\mathcal{V}}, \mathcal{P}$) to the ideal functionality $\mathcal{F}$(Ideal-NaiveKeyTheft).
- Simulator simP receives a message ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}, \mathcal{C}$): do nothing.

C.3.2. Indistinguishability of Real and Ideal Worlds. To prove indistinguishability of the real and ideal worlds from the perspective of the environment, we will go through a sequence of hybrid games.

Real world. We start with the real world with a dummy adversary that simply passes messages to and from the environment $\mathcal{E}$.

Hybrid 1. Hybrid 1 is the same as the real world, except that now the adversary (also referred to as a simulator) will call $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \mathsf{NIZK}.\widehat{\mathcal{K}}(1^{\lambda})$ to perform a simulated setup for the NIZK scheme. The simulator will pass the simulated $\widehat{\mathsf{crs}}$ to the environment $\mathcal{E}$. When an honest perpetrator $\mathcal{P}$ produces a NIZK proof, the simulator will replace the real proof with a simulated NIZK proof before passing it onto the

environment $\mathcal{E}$. The simulated NIZK proof can be computed by calling the $\mathsf{NIZK}.\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)$ algorithm which takes only the statement as input but does not require knowledge of a witness.

Fact 1. It is not hard to see that if the NIZK scheme is computational zero-knowledge, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 1 from the real world except with negligible probability.

Hybrid 2. The simulator simulates the $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$ functionality. Since all messages to the $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$ functionality are public, simulating the contract functionality is trivial. Therefore, Hybrid 2 is identically distributed as Hybrid 1 from the environment $\mathcal{E}$'s view.

Hybrid 3. Hybrid 3 is the same as Hybrid 2 except for the following changes. When an honest party sends a message to the contract (now simulated by the simulator $\mathcal{S}$), it will sign the message with a signature verifiable under an honestly generated nym. In Hybrid 3, the simulator will replace all honest parties' nyms and generate these nyms itself. In this way, the simulator will simulate honest parties' signatures by signing them itself. Hybrid 3 is identitally distributed as Hybrid 2 from the environment $\mathcal{E}$'s view.

Hybrid 4. Hybrid 4 is the same as Hybrid 3 except for the following changes. When the honest perpetrator $\mathcal{P}$ produces an ciphertext ct and if the contractor is also uncorrupted, then simulator will replace this ciphertext with an encryption of 0 before passing it onto the environment $\mathcal{E}$.

Fact 2. It is not hard to see that if the encryption scheme is semantically secure, then no polynomial-time environment $\mathcal{E}$ can distinguish Hybrid 4 from Hybrid 3 except with negligible probability.

Hybrid 5. Hybrid 5 is the same as Hybrid 4 except the following changes. Whenever the environment $\mathcal{E}$ passes to the simulator $\mathcal{S}$ a message signed on behalf of an honest party's nym, if the message and signature pair was not among the ones previously passed to the environment $\mathcal{E}$, then the simulator $\mathcal{S}$ aborts.

Fact 3. Assume that the signature scheme employed is secure, then the probability of aborting in Hybrid 5 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 5 would otherwise be identically distributed as Hybrid 4 modulo aborting.

Hybrid 6. Hybrid 6 is the same as Hybrid 5 except for the following changes. Whenever the environment passes ("claim", ct, π) to the simulator (on behalf of corrupted party $\mathcal{P}$), if the proof π verifies under the statement $(\mathsf{ct}, \mathcal{P})$, then the simulator will call the NIZK's extractor algorithm $\mathcal{E}$ to extract a witness $(r, \mathsf{sk}_{\mathcal{V}})$. If the NIZK π verifies under the statement $(\mathsf{ct}, \mathcal{P})$, and the extracted $\mathsf{sk}_{\mathcal{V}}$ does not satisfy $\mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}) = 1$, then abort the simulation.

Fact 4. Assume that the NIZK is simulation sound extractable, then the probability of aborting in Hybrid 6 is negligible. Notice that from the environment $\mathcal{E}$'s view, Hybrid 6 would otherwise be identically distributed as Hybrid 5 modulo aborting.

Finally, observe that Hybrid 6 is computationally indistinguishable from the ideal simulation $\mathcal{S}$ unless one of the following bad events happens:

- The $\mathsf{sk}_{\mathcal{V}}$ decrypted by an honest contractor $\mathcal{C}$ is different from that extracted by the simulator $\mathcal{S}$. However, given that the encryption scheme is perfectly correct, this cannot happen.
- The honest public key generation algorithm results in key collisions. Obviously, this happens with negligible probability if the encryption and signature schemes are secure.

Fact 5. Given that the encryption scheme is semantically secure and perfectly correct, and that the signature scheme is secure, then Hybrid 6 is computationally indistinguishable from the ideal simulation to any polynomial-time environment $\mathcal{E}$.

C.4. Extension to Incentive Compatible Key Theft Contract

Ideal program. The ideal program for an incentive compatible key theft contract is given in Figure 15.

Contract. The incentive compatible key theft contract is given in Figure 16 (a repeat of Figure 5 for the readers' convenience).

Protocol. The user-side programs for the incentive compatible key theft contract are supplied in Figure 17.

Theorem 3 (Incentive compatible key theft contract). Assume that the encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ is perfectly correct and semantically secure, the NIZK scheme is perfectly complete, computationally zero-knowledge and simulation sound extractable, then the protocol described in Figures 16 and 17 securely emulates $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$.

Proof: A trivial extension of the proof of Theorem 2, the naive key theft case.

Appendix D.
Formal Protocols for Public Document Leakage

D.1. Formal Description

Ideal program for public document leakage. We formally describe the ideal program for public document leakage in Figure 18.

Contract. The contract program for public leakage is formally described in Figure 19, which is a repeat of Figure 3 for the readers' convenience.

Protocol. The protocols for public leakage are formally described in Figure 20.

Theorem 4 (Public leakage). Assume that the encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ is perfectly correct and semantically secure, the NIZK scheme is perfectly complete and computationally zero-knowledge, then the protocol described in Figures 3 and 20 securely emulates $\mathcal{F}(\mathsf{Ideal\text{-}PublicLeaks})$.

Proof: The formal proofs are supplied in Appendix D.2.

D.2. Proofs for Public Document Leakage

D.2.1. Ideal World Simulator. The wrapper part of $\mathcal{S}(\mathsf{simP})$ was described earlier , we now describe the user-defined simulator simP.

Init. The simulator simP runs $\mathsf{crs} \leftarrow \mathsf{NIZK}.\mathcal{K}(1^\lambda)$, and $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}_{\mathsf{enc}}(1^\lambda)$. The simulator gives $(\mathsf{crs}, \mathsf{pk})$ to the environment $\mathcal{E}$, and remembers sk.

The simulator $\mathcal{S}(\mathsf{simP})$ will also simulate the random oracle (RO) queries. For now, we simply assume that a separate RO instance is employed for each protocol instance – or we can use the techniques by Canetti et al. [28] to have a global RO for all protocol instances.

Simulation for an honest seller $\mathcal{C}$.

- *Create:* Environment $\mathcal{E}$ sends input ("create", M, $\mathcal{C}$, T_{end}) to an honest leaker $\mathcal{C}$: simP receives ("create", $|M|$, $\mathcal{C}$) from the ideal functionality $\mathcal{F}(\mathsf{Ideal\text{-}PublicLeaks})$ – and this message is routed through $\mathcal{S}$. simP now generates an msk using the honest algorithm. For $i \in [n]$, pick $\mathsf{ct}_i \xleftarrow{\$} \{0,1\}^\ell$ where ℓ denotes the length of each document. Pick $c_0 := \mathsf{Enc}(\mathsf{pk}, 0, r_0)$ for some random r_0.
  Now, send ("create", c_0, T_{end}) to the internally simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$. Upon receiving a challenge set Ω from the ideal functionality, use the same Ω for simulating $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.
- *Confirm:* Upon receiving $\{m_i\}_{i\in\Omega}$ from the ideal functionality: the simulator simP now computes[3] $\kappa_i := \mathsf{PRF}(\mathsf{msk}, i)$ for $i \in \Omega$. The simulator programs the random oracle such that $H(\kappa_i) = m_i \oplus \mathsf{ct}_i$. Now, the simulator computes the NIZKs honestly, and send $\{\kappa_i, \pi_i\}_{i\in\Omega}$ to the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.
- *Accept:* Upon receiving ("accept", $\mathcal{P}$) from the ideal functionality, upon receiving M from the ideal functionality: send ("accept", msk) to the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$. Now, based on M, program the random oracle such that $H(\mathsf{PRF}(\mathsf{msk}, i)) \oplus \mathsf{ct}_i = m_i$ for $i \in [n]$.

Simulation for an honest purchaser $\mathcal{P}$.

- *Donate:* Environment sends ("donate", $\mathsf{amt}, $\mathcal{P}$) to an honest donor, simulator simP receives ("donate", $\mathsf{amt}, $\mathcal{P}$) from the ideal functionality (routed by the wrapper $\mathcal{S}$), and forwards it to the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.

Simulation for a corrupted purchaser $\mathcal{P}$.

- *Donate:* If the environment $\mathcal{E}$ sends ("donate", $\mathsf{amt}, $\mathcal{P}$) to simP on behalf of a corrupted purchaser $\mathcal{P}$ (message routed through the wrapper $\mathcal{S}$), simP passes it onto the ideal functionality, and the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.

Simulation for a corrupted leaker $\mathcal{C}$.

- *Create:* When the environment $\mathcal{E}$ sends ("create", (ct_0, $\{(i, \mathsf{ct}_i\}_{i\in[n]}$), T_{end}, $\mathcal{C}$) to simP, simP passes it to the internally simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$. Further, simP decrypts the msk from c_0.
  Now reconstruct M in the following manner: Compute all κ_i's from the msk. For every κ_i that was submitted as an RO query, the simulator recovers the m_i. Otherwise if for some i, κ_i was an RO query earlier, the simulator programs the RO randomly at κ_i, and computes the m_i accordingly – in this case m_i would be randomly distributed.
  Now, send ("create", M, T_{end}) on behalf of $\mathcal{C}$ to the ideal functionality where M is the document set reconstructed as above.
- *Challenge:* When the environment $\mathcal{E}$ sends ("confirm", $\{\kappa_i, \pi_i\}_{i\in\Omega}$, $\mathcal{C}$) to simP (message routed through the wrapper $\mathcal{S}$), pass the message to the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$. If the NIZK proofs all verify, then send "confirm" as $\mathcal{C}$ to the ideal functionality.
- *Accept:* When the environment $\mathcal{E}$ sends ("accept", msk, r_0, $\mathcal{C}$) to simP (message routed through the wrapper $\mathcal{S}$), pass the message to the simulated $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$. If $\mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0) = c_0$, then send "accept" as $\mathcal{C}$ to the ideal functionality.

Indistinguishability of real and ideal worlds. Given the above description of the ideal-world simulation, it is not hard to proceed and show the computational indistinguishability of the real and the ideal worlds from the perspective of the environment $\mathcal{E}$.

3. If the hash function has short output, we can compute the encryption of m_i as follows: $m_i \oplus [H(\kappa_i, 1, \text{"enc"}) \,||\, H(\kappa_i, 2, \text{"enc"}) \ldots, ||\, H(\kappa_i, z, \text{"enc"})]$ for suitably large z. Here we simply write $H(\kappa_i) \oplus m_i$ for convenience.

Appendix E.
Supplemental Details for Document Leakage

E.1. Background: Existing Darkleaks Protocol

In this appendix, we present an overview of the existing, broken Darkleaks protocol, as we are unaware of any unified technical presentation elsewhere. (Specific details, e.g., message formats, may be found in the Darkleaks source code [3], and cryptographic primitives h_1, h_2, h_2, and $(\mathsf{enc}, \mathsf{dec})$ are specified below.)

The protocol steps are as follows:

- *Create:* The contractor $\mathcal{C}$ partitions the secret $M = m_1 \parallel m_2 \parallel \ldots \parallel m_n$. For each segment m_i in $M = \{m_i\}_{i=1}^n$, $\mathcal{C}$ computes:
  - A Bitcoin (ECDSA) private key $\mathsf{sk}_i = h_1(m_i)$ and the corresponding public key pk_i.
  - The Bitcoin address $a_i = h_2(\mathsf{pk}_i)$ associated with pk_i.
  - A symmetric key $\kappa_i = h_3(\mathsf{pk}_i)$, computed as a hash of public key pk_i.
  - The ciphertext $e_i = \mathsf{enc}_{\kappa_i}[m_i]$.

  $\mathcal{C}$ publishes: The parameter triple (n, k, T_{open}), ciphertexts $E = \{e_i\}_{i=1}^n$, and Bitcoin addresses $A = \{a_i\}_{i=1}^n$.

- *Challenge:* At epoch (block height) T_{open}, the current Bitcoin block hash B_t serves as a pseudorandom seed for a challenge $S^* = \{s_i\}_{i=1}^k$.
- *Response:* In epoch T_{open}, $\mathcal{C}$ publishes the subset of public keys $PK^* = \{pk_s\}_{s \in S^*}$ corresponding to addresses $A^* = \{a_s\}_{s \in S^*}$. (The sample of segments $M^* = \{m_s\}_{s \in S^*}$ can then be decrypted by the Darkleaks community.)
- *Payment:* To pay for M, buyers send Bitcoin to the addresses $A - A^*$ corresponding to unopened segments.
- *Disclosure:* The leaker $\mathcal{C}$ claims the payments made to addresses in $A - A^*$. As spending the Bitcoin in address a_i discloses pk_i., *decryption of all unopened segments $M - M^*$ is automatically made possible for the Darkleaks community*.

Here, $h_1 =$ SHA-256, $h_2 =$ RIPEMD-160(SHA-256()), and $h_3 =$ SHA-256(SHA-256()). The pair $(\mathsf{enc}, \mathsf{dec})$ in Darkleaks corresponds to AES-256-ECB.

As a byproduct of its release of PK^* in response to challenge S^*, $\mathcal{C}$ proves (weakly) that undecrypted ciphertexts are well-formed, i.e., that $e_i = \mathsf{enc}_{\kappa_i}[m_i]$ for $\kappa_i = h_3(pk_i)$. This cut-and-choose-type proof assures buyers that when $\mathcal{C}$ claims its reward, M will be fully disclosed.

E.2. Public leakage implementation on Ethereum

The section illustrates an actual smart contract for public leakage. This contract fixes two main drawbacks with the existing Darkleaks protocol (Shortcomings 1 and 2 discussed in 4.1). The contract mainly enables better guarantees through deposits and timeout procedures, while preventing selective withholding. Figure 21 illustrates the contract code. The main goal of providing this code is to illustrate how fast it could be to write such contracts.

The contract in Figure 21 mainly considers a leaker who announces the ownership of the leaked material (e-mails, photos, secret documents, .. etc), and reveals a random subset of the encryption keys at some point to convince users of the ownership. Interested users can then deposit donations. In order for the leaker to get the reward from the contract, **all** the rest of the keys must be provided at the same time, before a deadline.

To ensure **incentive compatability**, the leaker is required by the contract in the beginning to deposit an amount of money, that is only retrievable if complied with the protocol. Also, for users to feel safe to deposit money, a timeout mechanism is used, such that if the leaker does not provide a response in time, the users will be able to withdraw the donations.

E.3. Private Secret-Leakage Contracts

In Section 4, we consider a public leakage model in which $\mathcal{C}$ collects donations, and when satisfied with total amount donated, leaks a secret to the public. In a variation in this appendix, we can consider a *private* leakage model in which $\mathcal{C}$ leaks a secret privately to a purchaser $\mathcal{P}$. A simple modification to the blackbox protocol supports this case. In particular, if $\mathcal{C}$ accepts $\mathcal{P}$'s bid, it computes the pair (ct, π) as follows:

- $\mathsf{ct} := \mathsf{Enc}(\mathsf{pk}_{\mathcal{P}}, \mathsf{msk}, r)$, for random coins r and where $\mathsf{pk}_{\mathcal{P}}$ denotes purchaser $\mathcal{P}$'s (pseudonymous) public key.
- π is a NIZK proof for the following statement:

$$\exists(\mathsf{msk}, r_0, r) \text{ s.t. } (c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)) \\ \wedge (\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_{\mathcal{P}}, \mathsf{msk}, r))$$

When $\mathcal{C}$ submits (ct, π) to the contract, the contract verifies the NIZK proof π, and if it is correct, sends $\mathcal{P}$'s deposited bid to $\mathcal{C}$. At this point, the purchaser $\mathcal{P}$ can decrypt the master secret key msk and then the unopened segments.

The above private leakage protocol can be proven secure in a similar manner as our public leakage contract.

A practical version for Ethereum. An efficient instantiation of this protocol is possible using a *verifiable random function* (VRF). and *verifiable encryption* (VE). We sketch the construction informally here (without proof). We then describe a specific pair of primitive choices (a VRF by Chaum and Pedersen [30] and VE by Camenisch and Shoup [25]) that can be efficiently realized in Ethereum.

Briefly, a VRF is a public-key primitive with private / public key pair $(\mathsf{sk}_{\text{vrf}}, \mathsf{pk}_{\text{vrf}})$ and an associated pseudorandom function F. It takes as input a value i and outputs a pair (σ, π), where $\sigma = F_{\mathsf{sk}_{\text{vrf}}}(i)$, and π is a NIZK proof of correctness of σ. The NIZK π can be verified using pk_{vrf}.

A VE scheme is also a public-key primitive, with private / public key pair $(\mathsf{sk}_{ve}, \mathsf{pk}_{ve})$. It takes as input a message m and outputs a ciphertext / proof pair (ct, π), where π is

a NIZK proof that $\mathsf{ct} = \mathsf{enc}_{\mathsf{pk}_{ve}}[m]$ for a message m that satisfies some publicly defined property θ.

Our proposed construction, then, uses a VRF to generate (symmetric) encryption keys for segments of M such that $\kappa_i = F_{\mathsf{sk}_{vrf}}(i)$. That is, $\mathsf{msk} = \mathsf{sk}_{vrf}$. The corresponding NIZK proof π is used in the **Confirm** step of the contract to verify that revealed symmetric keys are correct. A VE, then, is used to generate a ciphertext ct on $\mathsf{msk} = \mathsf{sk}_{vrf}$ under the public key $\mathsf{pk}_{\mathcal{P}}$ of the purchaser. The pair (ct, π), is presented in the **Accept** step of the contract. The contract can then verify the correctness of ct.

A simple and practical VRF due to Chaum and Pedersen [30] is one that for a group $\mathbb{G}$ of order p with generator g (and with some reasonable restrictions on p), $\mathsf{msk} = \mathsf{sk}_{vrf} = x$, for $x \in_R \mathbb{Z}_p$ and $\mathsf{pk}_{vrf} = g^x$. Then $F_{\mathsf{sk}_{\text{vrf}}}(i) = H(i)^x$ for a hash function $H : \{0,1\}^* \to \mathbb{G}$, while π is a Schnorr-signature-type NIZKP. (Security relies on the DDH assumption on $\mathbb{G}$ and the ROM for H.)

A corresponding, highly efficient VE scheme of Camenisch and Shoup [25] permits encryption of a discrete log over a group $\mathbb{G}$; that is, it supports verifiable encryption of a message x, where for a public value y, the property $\theta_{\mathbb{G}}(y)$ is $x = \mathsf{dlog}(y)$ over $\mathbb{G}$. Thus, the scheme supports verifiable encryption of $\mathsf{msk} = \mathsf{sk}_{vrf} = x$, where π is a NIZK proof that x is the private key corresponding to $\mathsf{pk}_{vrf} = g^x$. (Security relies on Paillier's decision composite residuosity assumption.)

Serpent, the scripting language for Ethereum, offers (beta) support for modular arithmetic. Thus, the Chaum-Pedersen VRF and Camensich-Shoup VE can be efficiently implemented in Ethereum, showing that private leakage contracts are practical in Ethereum.

Ideal-KeyTheft

Init: Set state := INIT.

Create: Upon recipient of ("create", \$reward, $\mathsf{pk}_{\mathcal{V}}, T_{\text{end}}, \Delta T$) from some contractor $\mathcal{C}$:
Same as Ideal-NaiveKeyTheft (Figure 12), and additionally store ΔT.

Intent: Upon recipient of ("intent", $\mathsf{sk}_{\mathcal{V}}$) from some perpetrator $\mathcal{P}$: Same as Ideal-NaiveKeyTheft.

Claim: Upon recipient of ("claim") from perpetrator $\mathcal{P}$:
Same as Ideal-NaiveKeyTheft except that the ledger update $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$ does not happen.

Revoke: Upon receiving ("revoke", Π_{revoke}) from some $\mathcal{R}$:
Notify $\mathcal{S}$ of ("revoke", Π_{revoke})
Assert Π_{revoke} is valid, and state $\neq$ ABORTED.
$\mathsf{ledger}[\mathcal{R}] := \mathsf{ledger}[\mathcal{R}] + \$\mathsf{smallreward}$.
If state = CLAIMED:
$t :=$ (time elapsed since successful **Claim**).
$\mathcal{P} :=$ (successful claimer).
$\mathsf{reward}_{\mathcal{P}} := f(\$\mathsf{reward}, t)$.
$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{reward}_{\mathcal{P}}$.
Else, $\mathsf{reward}_{\mathcal{P}} := 0$
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward} - \mathsf{reward}_{\mathcal{P}} - \$\mathsf{smallreward}$
Set state := ABORTED.

Timer: If state = CLAIMED and at least ΔT time elapsed since successful **Claim**:
$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$;
Set state := ABORTED.
Else if current time $T > T_{\text{end}}$ and state $\neq$ ABORTED:
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$.
Set state := ABORTED.

Figure 15: Thwarting revoke-and-claim attacks in the key theft ideal program.

Contract-KeyTheft

Init: Set state := INIT. Let crs := $\mathsf{KeyGen}_{\text{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string generated during a trusted setup process.

Create: Same as in Contract-NaiveKeyTheft (Figure 13), except that an additional parameter ΔT is additionally submitted by $\mathcal{C}$.

Intent: Same as Contract-NaiveKeyTheft.

Claim: Same as Contract-NaiveKeyTheft, except that the ledger update $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$ does not take place immediately.

Revoke: On receive ("revoke", Π_{revoke}) from some $\mathcal{R}$:

Assert Π_{revoke} is valid, and state $\neq$ ABORTED.
$\mathsf{ledger}[\mathcal{R}] := \mathsf{ledger}[\mathcal{R}] + \$\mathsf{smallreward}$.
If state = CLAIMED:
$t :=$ (time elapsed since successful **Claim**).
$\mathcal{P} :=$ (successful claimer)
$\mathsf{reward}_{\mathcal{P}} := f(\$\mathsf{reward}, t)$.
$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \mathsf{reward}_{\mathcal{P}}$.
Else, $\mathsf{reward}_{\mathcal{P}} := 0$
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward} - \$\mathsf{smallreward} - \mathsf{reward}_{\mathcal{P}}$
Set state := ABORTED.

Timer: If state = CLAIMED and at least ΔT time elapsed since successful **Claim**:

$\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$ where $\mathcal{P}$ is successful claimer;
Set state := ABORTED.
Else if current time $T > T_{\text{end}}$ and state $\neq$ ABORTED:
$\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$.
Set state := ABORTED.
// $\mathcal{P}$ should not submit claims after time $T_{end} - \Delta T$.

Figure 16: Key compromise CSC that thwarts revoke-and-claim attacks. Although supercially written in a slightly different manner, this figure is essentially equivalent to Figure 5 in the main body. We repeat it here and write the contract with respect to the differences from Figure 13 for the readers' convenience.

Prot-KeyTheft

Contractor $\mathcal{C}$:

Create: Upon receiving input ("create", $\$\mathsf{reward}$, $\mathsf{pk}_{\mathcal{V}}$, T_{end}, ΔT, $\mathcal{C}$):
Send ("create", $\$\mathsf{reward}$, $\mathsf{pk}_{\mathcal{V}}$, T_{end}, ΔT) to $\mathcal{G}$(Contract-KeyTheft).

Claim: Upon receiving a message ("claim", ct) from $\mathcal{G}$(Contract-KeyTheft):
Decrypt and output $m := \mathsf{Dec}(\mathsf{sk}_{\mathcal{C}}, \mathsf{ct})$.

Perpetrator $\mathcal{P}$:

Intent: Same as Prot-NaiveKeyTheft (Figure 14), but send messages to $\mathcal{G}$(Contract-KeyTheft) rather than $\mathcal{G}$(Contract-NaiveKeyTheft).

Claim: Same as Prot-NaiveKeyTheft, but send messages to $\mathcal{G}$(Contract-KeyTheft) rather than $\mathcal{G}$(Contract-NaiveKeyTheft).

Revoker $\mathcal{R}$:

Revoke: Upon receiving ("revoke", Π_{revoke}) from the environment $\mathcal{E}$: forward the message to $\mathcal{G}$(Contract-KeyTheft).

Figure 17: User-side programs for incentive compatible key theft.

Ideal-PublicLeaks

Init: Set state = INIT, and donations := {}.

Create: Upon receiving ("create", M, T_{end}) from some leaker $\mathcal{C}$, where M is a document consisting of n segments denoted $M := \{m_i\}_{i\in[n]}$:

Notify ("create", $|M|$, $\mathcal{C}$) to $\mathcal{S}$.

Select a random subset $\Omega \subset [n]$ of size k, and send Ω to the adversary $\mathcal{S}$.
Set state := CREATED.

Confirm: Upon receiving ("confirm") from leaker $\mathcal{C}$:

Assert state = CREATED.
Send $\{m_i\}_{i\in\Omega}$ to the adversary $\mathcal{S}$.
Set state := CONFIRMED.

Donate: Upon receiving ("donate", $amt) from some purchaser $\mathcal{P}$:

Notify $\mathcal{S}$ of ("donate", $amt, $\mathcal{P}$)

Assert state = CONFIRMED
Assert ledger[$\mathcal{P}$] $\geq$ $amt.
Set ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] − $amt.
donations := donations $\cup$ {($amt, $\mathcal{P}$)}.

Accept: Upon receiving ("accept") from $\mathcal{C}$:

Notify ("accept", $\mathcal{C}$) to the ideal adversary $\mathcal{S}$.

Assert state = CONFIRMED.
ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + sum(donations)
Send M to the ideal adversary $\mathcal{S}$.
Set state := ABORTED.

Timer: If state = CONFIRMED and $T > T_{\text{end}}$: $\forall$($amt, $\mathcal{P}$) $\in$ donations: let ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + $amt. Set state := ABORTED.

Figure 18: Ideal program for public leaks.

Contract-PublicLeaks

Init: Set state := INIT, and donations := {}. Let crs := $\mathsf{KeyGen}_{\text{nizk}}(1^\lambda)$, pk := $\mathsf{KeyGen}_{\text{enc}}(1^\lambda)$ denote hardcoded public parameters generated through a trusted setup.

Create: Upon receiving ("create", c_0, $\{\mathsf{ct}_i\}_{i=1}^n$, T_{end}) from some leaker $\mathcal{C}$:

Set state := CREATED.
Select a random subset $\Omega \subset [n]$ of size k, and send ("challenge", Ω) to $\mathcal{C}$.

Confirm: Upon receiving ("confirm", $\{(\kappa_i, \pi_i)\}_{i\in\Omega}$) from $\mathcal{C}$:

Assert state = CREATED.
Assert that $\forall i \in S$: π_i is a valid NIZK proof (under crs) for the following statement:

$$\exists(\mathsf{msk}, r_0),\ \text{s.t.}\ (c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)) \wedge (\kappa_i = \mathsf{PRF}(\mathsf{msk}, i))$$

Set state := CONFIRMED.

Donate: Upon receiving ("donate", $amt) from some purchaser $\mathcal{P}$:

Assert state = CONFIRMED.
Assert ledger[$\mathcal{P}$] $\geq$ $amt.
Set ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] − $amt.
donations := donations $\cup$ {($amt, $\mathcal{P}$)}.

Accept: Upon receiving ("accept", msk, r_0) from $\mathcal{C}$:

Assert state = CONFIRMED
Assert $c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$
ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] + sum(donations)
Send ("leak", msk) to all parties.
Set state := ABORTED.

Timer: If state = CONFIRMED and $T > T_{\text{end}}$: $\forall$($amt, $\mathcal{P}$) $\in$ donations: let ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + $amt. Set state := ABORTED.

Figure 19: A contract PublicLeaks that leaks a secret M to the public in exchange for donations. This figure is a repeat of Figure 3 for the readers' convenience.

Prot-PublicLeaks

Init: Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathsf{nizk}}(1^\lambda)$ and $\mathsf{pk} := \mathsf{KeyGen}_{\mathsf{enc}}(1^\lambda)$ denote hardcoded public parameters generated through a trusted setup.

As leaker $\mathcal{C}$:

Create: Upon receiving ("create", $M := \{m_i\}_{i\in[n]}, T_{\text{end}}, \mathcal{C}$) from the environment $\mathcal{E}$:

$\mathsf{msk} \xleftarrow{\$} \{0,1\}^\lambda$

For $i \in [n]$, compute $\kappa_i := \mathsf{PRF}(\mathsf{msk}, i)$. Then, compute $\mathsf{ct}_i := H(\kappa_i) \oplus m_i$.

Pick random $r_0 \xleftarrow{\$} \{0,1\}^\lambda$ and compute $c_0 := \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$.

Send ("create", c_0, $\{\mathsf{ct}_i\}_{i\in[n]}$, T_{end}). to $\mathcal{G}$(Contract-PublicLeaks).

Challenge: Upon receiving ("challenge", Ω) from $\mathcal{G}$(Contract-PublicLeaks):

For $i \in \Omega$: compute a NIZK proof π_i for the statement using witness (msk, r_0):

$$\exists(\mathsf{msk}, r_0), \text{ s.t. } (c_0 = \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)) \wedge (\kappa_i = \mathsf{PRF}(\mathsf{msk}, i))$$

Send ("confirm", $\{\kappa_i, \pi_i\}_{i\in\Omega}$). to $\mathcal{G}$(Contract-PublicLeaks).

Accept: Upon receiving ("accept", $\mathcal{C}$) from the environment: Send ("accept", msk, r_0). to $\mathcal{G}$(Contract-PublicLeaks).

As purchaser $\mathcal{P}$:

Donate: Upon receiving ("donate", \$amt, $\mathcal{P}$) from the environment $\mathcal{E}$: Send ("donate", \$amt). to $\mathcal{G}$(Contract-PublicLeaks).

Leak: Upon receiving ("leak", msk) from $\mathcal{G}$(Contract-PublicLeaks):

Download $\{(i, \mathsf{ct}_i)_{i\in[n]}\}$ from $\mathcal{G}$(Contract-PublicLeaks).

For $i \in [n]$, output $\mathsf{Dec}(H(\mathsf{PRF}(\mathsf{msk}, i)), \mathsf{ct}_i)$.

Figure 20: User-side programs for public leaks.

```
data leaker_address
data num_chunks
data revealed_set_size
data T_end
data deposit
data reveal_block_number
data selected_sample[]
data key_hashes[]
data donations[]
data sum_donations
data num_donors
data finalized

def init():
  self.leaker_address = msg.sender


# A leaker commits to the hashes of the
    encryption keys, and sets the announcement
    details
def commit( key_hashes:arr, revealed_set_size,
    reveal_block_number, T_end,
    distribution_address):
  # Assuming a deposit of a high value from the
    leaker to discourage aborting
  if( msg.value >= 1000000 and msg.sender == self
    .leaker_address and self.deposit == 0 and
    revealed_set_size < len(key_hashes)):
    self.deposit = msg.value
    self.num_chunks = len(key_hashes)
    self.revealed_set_size = revealed_set_size
    self.T_end = T_end
    self.reveal_block_number =
    reveal_block_number
    i  = 0
    while(i < len(key_hashes)):
      self.key_hashes[i] = key_hashes[i]
      i = i + 1
    return (0)
  else:
    return (-1)


def revealSample(sampled_keys:arr):
  # The contract computes and stores the random
    indices based on the previous block hash.
    The PRG is implemented using SHA3 here for
    simplicity.
  # The contract does not have to check for the
    correctness of the sampled keys. This can be
    done offline by the users.
  if( msg.sender == self.leaker_address and len(
    sampled_keys) == self.revealed_set_size and
    block.number == self.reveal_block_number ):
    seed = block.prevhash
    c = 0
    while(c < self.revealed_set_size):
      if(seed < 0):
        seed = 0 - seed
      idx = seed % self.num_chunks
      # make sure idx was not selected before
      while(self.selected_sample[idx] == 1):
        seed = sha3(seed)
        if(seed < 0):
          seed = 0 - seed
        idx = seed % self.num_chunks
      self.selected_sample[idx] = 1
      seed = sha3(seed)
      c = c + 1
    return(0)
  else:
    return(-1)
```

```
def donate():
  # Users verify the shown sample offline, and
    interested users donate money.
  prev_donation = self.donations[msg.sender]
  if( msg.value > 0 and block.timestamp <= self.
    T_end and prev_donation == 0):
    self.donations[msg.sender] = msg.value
    self.num_donors = self.num_donors + 1
    self.sum_donations = self.sum_donations + msg
    .value
    return(0)
  else:
    return(-1)


def revealRemaining(remaining_keys:arr):
  # For the leaker to get the reward, the
    remaining keys have to be all revealed at
    once.
  # The contract will check for the consistency
    of the hashes and the remaining keys this
    time.
  if( msg.sender == self.leaker_address and block
    .timestamp <= self.T_end and len(
    remaining_keys)==self.num_chunks - self.
    revealed_set_size and self.finalized == 0):
    idx1 = 0
    idx2 = 0
    valid = 1
    while(valid == 1 and idx1 < len(
    remaining_keys)):
      while(self.selected_sample[idx2] == 1):
        idx2 = idx2+1
      key = remaining_keys[idx1]
      key_hash = self.key_hashes[idx2]
      if(not(sha3(key) == key_hash)):
        valid = 0
      idx1 = idx1+1
      idx2 = idx2+1

    if(valid == 1):
      send(self.leaker_address, self.
    sum_donations + self.deposit)
      self.finalized = 1
      return (0)
    else:
      return(-1)
  else:
    return(-1)


def withdraw():
  ## This is a useful module that enables users
    to get their donations back if the leaker
    aborted
  v = self.donations[msg.sender]
  if(block.timestamp > self.T_end and self.
    finalized == 0 and v > 0):
    send(msg.sender, v + self.deposit/self.
    num_donors)
    return (0)
  else:
    return (-1)
```

Figure 21: Public leakage contract implemented on top of Ethereum.