The Ring of Gyges: Investigating the Future of Criminal Smart Contracts

Ari Juels
Cornell Tech (Jacobs)

Ahmed Kosba
Univ. of Maryland

Elaine Shi
Cornell Univ.

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... " —Plato, *The Republic*, Book 2 (2.360b) (trans. Benjamin Jowett)

Abstract

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

Next-generation cryptocurrencies such as Ethereum will include rich scripting languages in support of *smart contracts*, programs that autonomously intermediate transactions. In this paper, we explore the risk of smart contracts fueling 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 that CSCs for leakage of secrets (à la Wikileaks) are efficiently realizable in existing scripting languages such as that in Ethereum. We show that CSCs for theft of cryptographic keys can be achieved using 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 show similarly that authenticated data feeds, an emerging feature of smart contract systems, can facilitate CSCs for real-world crimes (e.g., property crimes).

Our results highlight the urgency of creating policy and technical safeguards against CSCs in order to realize the 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 for many applications, these features have a dark side. Bitcoin has stimulated the growth of ransomware [6], money laundering [40], and illicit commerce, as exemplified by the notorious Silk Road [32].

New cryptocurrencies such as Ethereum (as well as systems such as Counterparty [48] and SmartContract [1]) 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 successfully delivered data or services. Given the flexibility of such smart contract systems, it is to be expected that they will stimulate not just new beneficial services, but 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 (confidential) delivery of an target key sk, such as a certificate authority's private digital signature key.

We explore the following key questions in this paper. *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 conventional online systems?* Exploring these questions is essential to identifying threats and devising countermeasures.

1.1 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 achieves a key property in this paper that we call *commission-fair*, meaning informally that its execution guarantees *both* commission of a crime and commensurate payment for the perpetrator of the crime or *neither*. (We formally define commission-fairness for individual CSCs in the paper.) Fair exchange is necessary to ensure commission-fairness, but not sufficient: We show how CSC constructions implementing fair exchange still allow a party to a CSC to cheat. Correct construction of CSCs can thus be quite delicate.

Second, even if a CSC can in principle be constructed, given the limited opcodes 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 execution fees levied against the CSC.

The following example illustrates these challenges.

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) CertoMart. $\mathcal{C}$ offers a reward $\$\mathsf{reward}$ to a perpetrator $\mathcal{P}$ for (confidentially) delivering the CertoMart private key $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{C}$.

To ensure fair exchange of the key and reward in Bitcoin, $\mathcal{C}$ and $\mathcal{P}$ would need to use a trusted third party or communicate directly, raising the risks of being cheated or discovered by law enforcement. They could vet one another using a reputation system, but such systems are often infiltrated by law enforcement authorities [57]. In contrast, a decentralized smart contract can achieve self-enforcing fair exchange. For key theft, this is possible using the CSC Key-Theft in the following example:

Example 1b (Key compromise CSC) *$\mathcal{C}$ generates a private / public key pair $(\mathsf{sk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{C}})$ and initializes* Key-Theft *with public keys $\mathsf{pk}_{\mathcal{C}}$ and $\mathsf{pk}_{\mathcal{V}}$ (the CertoMart public key).* Key-Theft *awaits input from a claimed perpetrator $\mathcal{P}$ of a pair (ct, π), where π is a zero-knowledge proof that $\mathsf{ct} = enc_{\mathsf{pk}_{\mathcal{C}}}[sk_{\mathcal{V}}]$ is well-formed.* Key-Theft *then verifies π and upon success sends a reward of* $\$\mathsf{reward}$ *to $\mathcal{P}$. The contractor $\mathcal{C}$ can then download and decrypt* ct *to obtain the compromised key $\mathsf{sk}_{\mathcal{V}}$.*

Key-Theft implements a fair exchange between $\mathcal{C}$ and $\mathcal{P}$, paying a reward to $\mathcal{P}$ *if and only if* $\mathcal{P}$ delivers a valid key (as proven by π), eliminating the need for a trusted third party. But it is *not commission-fair*, as it does not ensure that sk_{vict} actually has value. The CertoMart can neutralize the contract by preemptively revoking its own certificate and then itself claiming $\mathcal{C}$'s reward $\$\mathsf{reward}$!

As noted, a major thrust of this paper is showing how, for CSCs such as Key-Theft, criminals will be able to bypass such problems and still construct commission-fair CSCs. (For key compromise, it is necessary to enable contract cancellation should a key be revoked.) Additionally, we show that these CSCs can be efficiently realized using existing cryptocurrency tools or features currently envisioned for cryptocurrencies (e.g., zk-SNARKS [20]).

1.2 This paper

We show that it is or will be possible in smart contract systems to construct commission-fair CSCs for three types of crime:

1. Leakage / sale of secret documents;
2. Theft of private keys; and
3. "*Calling-card*" crimes, a broad class of physical-world crimes (murder, arson, etc.)

The fact that CSCs are possible in principle is not surprising. Previously, however, it was not clear how practical or extensively applicable CSCs might be. As our constructions for commission-fair CSCs show, constructing CSCs is not as straightforward as it might seem, but new cryptographic techniques and new approaches to smart contract design can render them feasible and even practical. Furthermore, criminals will undoubtedly devise CSCs beyond what this paper and the community in general are able to anticipate.

Our work therefore shows how imperative it is for the community to consider the construction of defenses against CSCs. 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 [14] and the launch of Ethereum took place in July 2015. *It is only by recognizing CSCs early in their lifecycle that the community can develop timely countermeasures to them,* and see the promise of distributed smart contract systems fully realized.

While our focus is on preventing evil, happily the techniques we propose can also be used to create beneficial 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 beneficial smart contracts, CSC construction requires a careful combination of cryptography with commission-fair design [35].

In summary, our contributions are:

- *Criminal smart contracts:* We initiate the study of CSCs as enabled by Turing-complete scripting languages 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 data sources called "authenticated data feeds" (described below) 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.

  We emphasize that because commission-fairness means informally that contracting parties obtain their "expected" utility, an application-specific metric, commission-fairness must be defined in a way specific to a given CSC. We thus formally specify commission-fairness for each of our CSC constructions in the relevant paper appendices.
- *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.
- *Countermeasures:* We briefly discuss in Section 7 some possible approaches to designing smart contract systems with countermeasures against CSCs. While this discussion is preliminary, a key contribution of our work is to show the need for such countermeasures and stimulate exploration of their implementation in smart contract systems such as Ethereum.

We also briefly discuss in Appendix B how maturing technologies, such as hardware roots of trust (e.g., Intel SGX [43]) and program obfuscation can enrich the space of possible CSCs—as they can, of course, beneficial smart contracts.

2 Background and Related Work

Emerging decentralized cryptocurrencies [55, 63] 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 supports 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 [21, 15, 7, 56, 49].

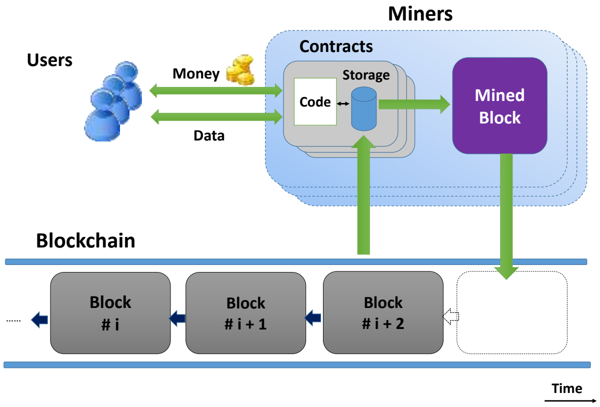

Figure 1: **Schematic of a decentralized cryptocurrency system with smart contracts**, as illustrated by Delmolino et al. [35]. A smart contract's state is stored on the public blockchain. A smart contract program is executed by a network of miners who reach consensus on the outcome of the execution, 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.

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 recently launched Ethereum [63]. Informally, a smart contract in such a system may be thought of as an autonomously executing piece of code whose inputs and outputs can include money. (We give more formalism below.) Hobbyists and companies are already building atop or forking off Ethereum to develop various smart contract applications such as security and derivatives trading [48], prediction markets [5], supply chain provenance [11], 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, a majority of the miners (as measured by computational resources) 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 user side, with no loss in security.

2.1 Smart contracts: the good and bad

Decentralized smart contracts have many beneficial uses, including the realization of a rich variety of new financial instruments. 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 these 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, 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. These are in their infancy in Ethereum, but their availability is growing.

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 [57, 41].
- *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 can set up a contract and walk away, allowing it to execute autonomously with no further interaction.
- *Enriched transactions with external state* broaden the scope of possible CSCs to, 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.

2.2 Digital cash and crime

Bitcoin and smart contracts do not represent the earliest emergence of cryptocurrency. Anonymous e-cash was introduced in 1982 in a seminal paper by David Chaum [30]. Naccache and von Solms noted that anonymous currency would render "perfect crimes" such as kidnapping untraceable by law enforcement [61]. This observation prompted the design of fair blind signatures or "escrow" for e-cash [24, 62], 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, but not pseudonymous cryptocurrencies such as Bitcoin.

Ransomware has appeared in the wild since 1989 [16]. A major cryptovirological [64] "improvement" to ransomware has been use of Bitcoin [47], thanks to which CryptoLocker ransomware has purportedly netted hundreds of millions of dollars in ransom [23]. Assassination markets using anonymous digital cash were first proposed in a 1995-6 essay entitled "Assassination Politics" [17].

There has been extensive study of Bitcoin-enabled crime, such as money laundering [54], Bitcoin theft [52], and illegal marketplaces such as the Silk Road [32]. Meiklejohn et al. [52] 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 (Ross Ulbricht), the operator of the Silk Road, even after seizing his laptop [59]. Möser, Böhome, and Breuker [54] 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 cryptocurrencies. Increasingly practical proposals to use NIZK proofs for anonymity in cryptocurrencies [18, 34, 53], some planned for commercial deployment, promise to make stronger anonymity available to criminals.

3 Notation and Threat Model

We adopt the formal blockchain model proposed by Kosba et al. [45]. 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. Our 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. A contract interacts with users and other contracts 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). Contracts can access and update the ledger to implement money transfers between users, who are represented by pseudonymous public keys.

3.1 Threat Model

We adopt the following threat model in this paper.

- *Blockchain: Trusted for correctness but not privacy.* We assume that the blockchain always correctly stores data and performs computations and is always available. The blockchain exposes all of its internal states to the public, however, 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 benefit. 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 between the blockchain and players are delivered within a bounded delay, i.e., not permanently dropped. (A player can always resend a transaction dropped by a malicious miner.) In our model, an adversary immediately receives and can arbitrarily reorder messages, however. In real-life decentralized cryptocurrencies, the winning miner determines the order of message processing. An adversary may collude with certain miners or influence message-propagation among nodes. As we show in Section 5, for key-theft contracts, message-reordering enables a rushing attack that a commission-fair CSC must prevent.

The formal model we adopt (reviewed later in this section and described in full by Kosba et al. [45]) captures all of the above aspects of our threat model.

3.2 Security definitions

For a CSC to be commission-fair requires two things:

- *Correct definition of commission-fairness.* There is no universal formal definition of commission fairness: It is application-specific, as it depends on the goals of the criminal (and perpetrator). Thus, for each CSC, we specify in the paper appendix a corresponding definition of commission-fairness by means of a UC-style ideal functionality that achieves it. Just specifying a correct ideal functionality is itself often challenging! We illustrate the challenge in Section 5 and Appendix D with a naive-key functionality that represents seemingly correct but in fact flawed key-theft contract.
- *Correct protocol implementation.* To prove that a CSC is commission-fair, we must show that its (real-world) protocol emulates the corresponding ideal functionality. We prove this for our described CSCs 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. Our protocols are also secure against aborting adversaries, e.g., attempts to abort without paying the other party. Fairness in the presence of aborts is well known in general to be impossible in standard models of distributed computation [33]. Several recent works, show that a blockchain that is correct, available, and aware of the progression of time can enforce financial fairness against aborting parties [21, 45, 15]. Specifically, when a contract lapses, the blockchain can cause the aborting party to lose a deposit to the honest parties.

3.3 Notational Conventions

We now explain some notational conventions for writing contracts. Appendix A gives a warm-up example.

- **Currency and ledger.** We use 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. The formal blockchain model [45] we adopt provides a contract wrapper 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 $\mathcal{P}$.

This notational system [45] is not only designed for convenience, but is also endowed with precise, formal meanings compatible with the Universal Composability framework [26]. We refer the reader to [45] for formal modeling details. While our proofs in the paper appendices rely on this supporting formalism, the main body can be understood without it.

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 deployed over Bitcoin can be made more robust and functionally enhanced as a smart contract and can be practically implemented in 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] (a kind of subsidized Wikileaks) serves as a decentralized market for crowdfunded 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.]."

Intuitively, we define commission-fairness in this setting to mean that a contractor $\mathcal{C}$ receives payment iff it leaks a secret in its entirety within a specified time limit. (See Appendix E for a formal definition.) As we show, Darkleaks highlights the inability of Bitcoin to support commission-fairness. We show how a CSC can in fact 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 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 F.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]

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 selected 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 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 basic 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 victimized 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 [38].)

[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 [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: In *any* protocol in which goods are represented by a random sample, not just Darkleaks, $\mathcal{C}$ can insert a small number of valueless segments into M. With non-negligible probability, these will not appear in the sample Ω, so Ω necessarily provides only a *weak* guarantee of the global validity of M. The larger k and n, the smaller the risk of such attack.

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 settings where $\mathcal{C}$ aims to sell M for public release after revealing sample 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 narrow 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 some well known randomness source, e.g., the NIST randomness beacon [9].
- *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 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 CSC PublicLeaks for implementing this public leakage protocol is given in Figure 2. The corresponding user side is as explained informally above (and inferable from the contract).

```
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", $c_0$, $\{\mathsf{ct}_i\}_{i=1}^n$, $T_{\text{end}}$) from some leaker $\mathcal{C}$:
             Set state := CREATED.
             Select a random subset $\Omega \subset [n]$ of size $k$, and send ("challenge", $\Omega$) to $\mathcal{C}$.
Confirm: Upon receiving ("confirm", $\{(\kappa_i, \pi_i)\}_{i\in\Omega}$) from $\mathcal{C}$:
             Assert state = CREATED.
             Assert that $\forall i \in S$: $\pi_i$ is a valid NIZK proof (under crs) for the following statement:
                 $\exists(\mathsf{msk}, r_0)$, 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 2: A contract PublicLeaks that leaks a secret M to the public in exchange for donations.

4.3 Commission-fairness: Formal definition and proof

In Appendix E, we give a formal definition of commission-fairness for public leakage (explained informally above) as an ideal functionality. We also prove that PublicLeaks realizes this functionality assuming all revealed segments are valid—a property enforced with high (but not overwhelming) probability by random sampling of M in PublicLeaks.

4.4 Optimizations and Ethereum implementation

The formally specified contract PublicLeaks uses generic cryptographic primitives in a black-box manner. We now give a practical, optimized version, relying on the random oracle model (ROM), that eliminates trusted setup, and also achieves better efficiency and easy integration with Ethereum [63].

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

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

The master secret key is simply $\mathsf{msk} := \{\kappa_1, \ldots, \kappa_n\}$, i.e., the set of hash pre-images. As in PublicLeaks, 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. To accept donations, $\mathcal{C}$ reveals the entire msk.

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

Ethereum-based implementation. To demonstrate the feasibility of implementing leakage contracts using currently available technology, we implemented a version of the contract PublicLeaks atop Ethereum [63], using the Serpent contract language [10]. We specify the full implementation in detail in Appendix F.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, again, Darkleaks recommends $n = 100$. (A hash function call in practice takes a few micro-seconds, e.g., **3.92 μsecs** measured on a core i7 processor.)

4.5 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 F.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 black-box 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 [31] (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 support for big number arithmetic in Ethereum.

5 A Key-Compromise CSC

Example 1b in the paper introduction 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 $pk_\mathcal{V}$. Recall that $\mathcal{C}$ 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 $\mathsf{pk}_\mathcal{V}$.

Intuitively, a key-theft contract is commission-fair if it rewards a perpetrator $\mathcal{P}$ for delivery of a private key that: (1) $\mathcal{P}$ was responsible for stealing and (2) Is valid for a substantial period of time. (We formally define it in Appendix D.)

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 could solicit abuse, but not full compromise of a private key, e.g., forged certificates.)

Figure 3 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

```
Contract KeyTheft-Naive
  Init:   Set state := INIT. Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathrm{nizk}}(1^\lambda)$ denote
          a hard-coded NIZK common reference string gener-
          ated during a trusted setup process.
  Create: Upon receiving ("create", $reward, $\mathsf{pk}_\mathcal{V}$, $T_{\mathrm{end}}$) from
          some contractor $\mathcal{C} := (\mathsf{pk}_\mathcal{C}, \ldots)$:
              Assert state = INIT.
              Assert $\mathsf{ledger}[\mathcal{C}] \geq \$\mathsf{reward}$.
              $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] - \$\mathsf{reward}$.
              Set state := CREATED.
  Claim:  Upon receiving ("claim", ct, $\pi$) from some purported
          perpetrator $\mathcal{P}$:
            Assert state = CREATED.
            Assert that $\pi$ is a valid NIZK proof (under crs) for
            the following statement:
                $\exists r, \mathsf{sk}_\mathcal{V}$ s.t. $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_\mathcal{C}, (\mathsf{sk}_\mathcal{V}, \mathcal{P}), r)$
                  and $\mathsf{match}(\mathsf{pk}_\mathcal{V}, \mathsf{sk}_\mathcal{V}) = \mathtt{true}$
            $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$.
            Set state := CLAIMED.
  Timer:  If state = CREATED and current time $T > T_{\mathrm{end}}$:
            $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$
            state := ABORTED
```

Figure 3: A naïve, flawed key theft contract (lacking commission-fairness)

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, 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 CA $\mathcal{V}$ can revoke the key $\mathsf{sk}_\mathcal{V}$ and then itself submit the key 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 the delivery of a *usable* private key sk_V.

Rushing attack. Another attack is a rushing attack. As noted in Section 3, an adversary can arbitrarily re-order messages—a reflection of possible attacks against the network layer in a cryptocurrency. (See also the formal blockchain model [45].) Thus, given a valid claim from perpetrator $\mathcal{P}$, a corrupt $\mathcal{C}$ can decrypt and learn $\mathsf{sk}_\mathcal{V}$, construct another valid-looking claim of its own, and make its own claim arrive before the valid one.

5.2 Fixing flaws in KeyTheft-Naive

We now show how to modify KeyTheft-Naive to prevent the above two attacks and achieve commission-fairness.

Thwarting revoke-and-claim attacks. 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 claim the reward. The result is that $\mathcal{C}$ pays \$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 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 $\mathsf{pk}_\mathcal{V}$.

$\mathcal{C}$ could submit Π_{revoke}, but to minimize interaction by $\mathcal{C}$, KeyTheft could provide a reward \$smallreward to a third-party submitter. The reward could be small, as Π_{revoke} would be easy for ordinary users 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 ($T_{\mathrm{revoke}} = \infty$ if no revocation happens prior to T_{end}). Then the contract assigns to $\mathcal{P}$ a reward of $f(\mathsf{reward}, t)$, where $t = \min(T_{\mathrm{end}}, T_{\mathrm{revoke}}) - T_{\mathrm{claim}}$.

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 [22].) 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 rushing attacks. To thwart rushing attacks, we separate the claim into two phases. In the first phase, $\mathcal{P}$ expresses an intent to claim by submitting a commitment of the real claim message. $\mathcal{P}$ then waits for the next round to open the commitment and reveal 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 claim the commitment to any string s.) In real-life decentralized cryptocurrencies, $\mathcal{P}$ can potentially wait multiple block intervals before opening the commitment, to have higher confidence that the blockchain will not fork. In our formalism, one round can correspond to one or more block intervals.

Figure 4 gives a key theft contract KeyTheft that thwarts revoke-and-claim and the rushing attacks.

5.3 Target and state exposure

An undesirable property of KeyTheft-Naive is that its target / victim and state are publicly visible. $\mathcal{V}$ can thus learn whether it is the target of KeyTheft-Naive. $\mathcal{V}$ also observes successful claims—i.e., whether $\mathsf{sk}_{\mathcal{V}}$ has been stolen—and can thus take informed defensive action. For example, as key revocation is expensive and time-consuming, $\mathcal{V}$ might wait until a successful claim occurs and only then perform a revoke-and-claim attack.

To limit target and state exposure, wenote 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 what we call *cover claims*, false claims that conceal any true claim. Our implementation of KeyTheft, as specified in Figure 4, 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 [39]).

A challenge here is that the contract state is public, thus the contract must be able to verify the proof for a valid claim (private key) $\mathsf{sk}_{\mathcal{V}_i}$ *without knowing which key was furnished, i.e., without learning i*. Our implementation shows that constructing such proofs as zk-SNARKs is practical. (The contractor $\mathcal{C}$ itself can easily learn i by decrypting $\mathsf{sk}_{\mathcal{V}_i}$, generating $\mathsf{pk}_{\mathcal{V}_i}$, and identifying 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, it is possible instead to 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.

```
Contract KeyTheft
    Init: Set state := INIT. Let crs := $\mathsf{KeyGen}_{\mathsf{nizk}}(1^\lambda)$ denote a hard-coded NIZK common reference string
          generated during a trusted setup process.
  Create: Same as in Contract KeyTheft-Naive (Figure 3), except that an additional parameter $\Delta T$ 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 cm, $\mathcal{P}$
   Claim: Upon receiving ("claim", ct, $\pi$, $r$) 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", $\Pi_{\text{revoke}}$) from some $\mathcal{R}$:
              Assert $\Pi_{\text{revoke}}$ 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 $\Delta T$ 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 4: Key compromise CSC that thwarts the revoke-and-claim attack and the rushing attack.

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. So that $\mathcal{C}$ need not interact with the contract after creation, the contract could parcel out small rewards at time T_{end} to third parties that submit cover claims. We do not implement cover claims in our version of KeyTheft nor include them in Figure 4.s

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 Commision-fairness: Formal definition and proof

We define commission-fairness for key theft in terms of an ideal functionality in Appendix D and also provide a formal proof of security there for KeyTheft.

5.5 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 use a generic transformation described in the Hawk work [45] to lift 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.) In our implementation, we assume the key generation is executed confidentially by a trusted party; otherwise a prover can produce a valid proof for a false statement. To minimize trust in the key generation phase, secure multi-party computation techniques can be used as in [19].

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 [44] 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 [20] 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 [4] 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 implementation 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 [12] and the CRL last updated at [8] on Feb 15th, 2016, that had a size of 143KB. On average, the verification executed in about 0.016 seconds on a 2.3 GHz i7 processor. The signature algorithm was SHA-256 with RSA encryption, 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 for single-target contracts is straightforward: The contract can determine whether a revocation proof corresponds to the targeted key. In multi-target contracts, though, the contract does not know which target key corresponds to the proof of key theft $\mathcal{P}$ submitted. Thus, a proof is needed that the revocation corresponds to the stolen key, and it must be submitted by $\mathcal{C}$.

We built a zk-SNARK circuit through which $\mathcal{C}$ can prove the connection between the ciphertext submitted

[2] The circuit also has other signature and encryption sub-circuits needed for simulation extractability – see Appendix C.3.

by the perpetrator and a target key with a secret index. For efficiency, we eliminated the need for the key-check sub-circuit 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, decentralized smart contract systems (e.g., Ethereum) have supporting services that provide authenticated data feeds, digitally signed attestations to news, facts about the physical world, etc. While still in its infancy, this powerful capability is fundamental to many applications of smart contracts and will expand 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* Assassinate *for the assassination of Senator X. The contract rewards the perpetrator $\mathcal{P}$ of this crime.*

The contract Assassinate *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,* Assassinate *searches an authenticated data feed on current events 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). Calling cards, alongside authenticated data feeds, can support 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 $\mathcal{P}$.

Intuitively, we define commission fairness to mean that $\mathcal{P}$ receives a reward iff it was responsible for carrying out a commissioned crime. (A formal definition is given in Appendix H.)

In more detail, 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 of pairs 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 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 once created, 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 a feed 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.

As example parameterization, we might let $\mathsf{CC} = \{0,1\}^{256}$, i.e., cc is a 256-bit string. A perpetrator $\mathcal{P}$ simply selects a calling card $\mathsf{cc} \xleftarrow{\$} \{0,1\}^{256}$ and commitment $\mathsf{vcc} := \mathsf{commit}(\mathsf{cc}, \mathcal{P}; \rho)$, where commit denotes a commitment scheme, and $\rho \in \{0,1\}^{256}$ a random string. (In practice, HMAC-SHA256 is a suitable choice for easy implementation in Ethereum, given its support for SHA-256.) $\mathcal{P}$ decommits by revealing all arguments to commit.

The CSC SiteDeface is shown in Figure 5.

Remarks. SiteDeface could be implemented alterna-

```
Contract SiteDeface
  Init: On receiving ($reward, $\mathsf{pk}_{\mathcal{S}}$, url, stmt) from some $\mathcal{C}$:
      Store ($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 $\mathsf{vcc}_i = \mathsf{commit}(\mathsf{cc}, \mathcal{P}; \rho)$, abort if not 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 $reward to $P_i$ and abort.
```

Figure 5: CSC for website defacement

tively 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. See Appendix G.

Implementation. Given an authenticated data feed, implementing SiteDeface would be straightforward and efficient. The main overhead lies in the **Claim** module, where the contract computes a couple of hashes and validates the feed signature on retrieved website data. As noted 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 needs only about 10msec to verify an RSA-2048 signature.

6.2 Commission-fairness: Formal definition

We give a formal definition of commission-fairness for a general calling-card CSC in Appendix H. We do not provide a security proof, as this would require modeling of physical-world systems, which is outside the scope of this paper.

6.3 Other calling-card crimes

Using a CSC much like SiteDeface, a contractor $\mathcal{C}$ can solicit many other crimes, e.g., assassination, assault, sabotage, hijacking, kidnapping, denial-of-service attacks, and terrorist attacks. 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.) We discuss these issues in Appendix G.

7 Countermeasures

The main aim of our work is to emphasize the importance of research into countermeasures against CSCs for emerging smart contract systems such as Ethereum. We briefly discuss this challenge and one possible approach.

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. A proactive alternative noted in Section 2 is an identity-escrow idea in early (centralized) e-cash systems sometimes referred as "trustee-based tracing" [24, 62]. Trustee-tracing schemes permitted a trusted party ("trustee") or a quorum of such parties to trace monetary transactions that would otherwise remain anonymous. In decentralized cryptocurrencies, however, users do not register identities with authorities—and many would object to doing so. It would be possible for users to register voluntarily with authorities of their choice, and for users to choose only to accept only currency they deem suitably registered. The notion of tainting coins, however, has been poorly received by the cryptocurrency community because it undermines the basic cash-like property of fungibility [13, 51], and trustee-based tracing would have a similar drawback. It is also unclear what entities should be granted the authority to perform blacklisting or register users.

We propose instead the notion of *trustee-neutralizable smart contracts*. A smart contract system might be designed such that an authority, quorum of authorities, or suitable set of general system participants 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, as are the right supporting technical mechanisms. We believe, however, that such a countermeasure might prove easier to implement than blacklisting or user registration.

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 support in smart contract systems such as Ethereum. The contract PublicLeaks and its private variant can today be efficiently implemented in Serpent, an Ethereum scripting language. KeyTheft would require only modest, already envisioned opcode support for zk-SNARKs for efficient deployment. Calling-card CSCs will be possible given a sufficiently rich data-feed ecosystem. Many more CSCs are no doubt possible.

We emphasize that smart contracts in distributed cryptocurrencies have numerous promising, legitimate applications and that 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 of such smart contracts while supporting their many powerful, beneficial applications.

References

[1] http://www.smartcontract.com.

[2] http://koinify.com.

[3] https://github.com/darkwallet/darkleaks.

[4] Amazon EC2 pricing. http://aws.amazon.com/ec2/pricing/.

[5] Augur. http://www.augur.net/.

[6] Bitcoin ransomware now spreading via spam campaigns. http://www.coindesk.com/bitcoin-ransomware-now-spreading-via-spam-campaigns/.

[7] bitoinj. https://bitcoinj.github.io/.

[8] CRL issued bby Symantec Class 3 EV SSL CA - G3. http://ss.symcb.com/sr.crl.

[9] NIST randomness beacon. https://beacon.nist.gov/home.

[10] Serpent. https://github.com/ethereum/wiki/wiki/Serpent.

[11] Skuchain. http://www.skuchain.com/.

[12] Verisign revoked certificate test page. https://test-sspev.verisign.com:2443/test-SPPEV-revoked-verisign.html. Accessed: 2015-05-15.

[13] Mt. Gox thinks it's the Fed. freezes acc based on "tainted" coins. (unlocked now). https://bitcointalk.org/index.php?topic=73385.0, 2012.

[14] Ethereum and evil. Forum post at Reddit; url: http://tinyurl.com/k8awj2j, Accessed May 2015.

[15] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure Multiparty Computations on Bitcoin. In *S & P*, 2013.

[16] J. Bates. Trojan horse: AIDS information introductory diskette version 2.0,. In E. Wilding and F. Skulason, editors, *Virus Bulletin*, pages 3–6. 1990.

[17] J. Bell. Assassination politics. http://www.outpost-of-freedom.com/jimbellap.htm, 1995-6.

[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, M. Green, E. Tromer, and M. Virza. Secure sampling of public parameters for succinct zero knowledge proofs. In *S & P*, 2015.

[20] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, 2014.

[21] I. Bentov and R. Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.

[22] L. Bilge and T. Dumitras. Before we knew it: an empirical study of zero-day attacks in the real world. In *CCS*, 2012.

[23] V. Blue. Cryptolocker's crimewave: A trail of millions in laundered Bitcoin. *ZDNet*, 22 December 2013.

[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, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *Theory of Cryptography*, pages 61–85. Springer, 2007.

[28] R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In *CCS*, 2014.

[29] R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO*, 2003.

[30] D. Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1983.

[31] D. Chaum and T. P. Pedersen. Wallet databases with observers. In *CRYPTO'92*, pages 89–105, 1993.

[32] N. Christin. Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace. In *WWW*, 2013.

[33] R. Cleve. Limits on the security of coin flips when half the processors are faulty. In *STOC*, 1986.

[34] G. Danezis, C. Fournet, M. Kohlweiss, and B. Parno. Pinocchio Coin: building Zerocoin from a succinct pairing-based proof system. In *PETShop*, 2013.

[35] 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.

[36] S. Egelman, C. Herley, and P. C. van Oorschot. Markets for zero-day exploits: Ethics and implications. In *NSPW*. ACM, 2013.

[37] P. T. et al. Darkwallet on twitter: "DARK LEAKS coming soon. http://t.co/k4ubs16scr". Reddit: http://bit.ly/1A9UShY.

[38] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *FC*, 2014.

[39] E. F. Foundation. EFF SSL observatory. URL: https://www.eff.org/observatory, August 2010.

[40] A. Greenberg. 'Dark Wallet' is about to make Bitcoin money laundering easier than ever. http://www.wired.com/2014/04/dark-wallet/.

[41] A. Greenberg. Alleged silk road boss Ross Ulbricht now accused of six murders-for-hire, denied bail. *Forbes*, 21 November 2013.

[42] J. Groth. Simulation-sound nizk proofs for a practical language and constant size group signatures. In *ASIACRYPT*, pages 444–459, 2006.

[43] Intel. Intel software guard extensions programming reference. Whitepaper ref. 329298-002US, October 2014.

[44] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1, 2003. RFC 3447.

[45] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. 2016.

[46] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papamanthou, R. Pass, abhi shelat, and E. Shi. How to use snarks in universally composable protocols. https://eprint.iacr.org/2015/1093.pdf, 2015.

[47] V. Kotov and M. Rajpal. Understanding crypto-ransomware. Bromium whitepaper, 2014.

[48] A. Krellenstein, R. Dermody, and O. Slama. Counterparty announcement. https://bitcointalk.org/index.php?topic=395761.0, January 2014.

[49] R. Kumaresan and I. Bentov. How to Use Bitcoin to Incentivize Correct Computations. In *CCS*, 2014.

[50] P. Mateus and S. Vaudenay. On tamper-resistance from a theoretical viewpoint. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 411–428. 2009.

[51] J. Matonis. Why Bitcoin fungibility is essential. *CoinDesk*, 1 Dec. 2013.

[52] 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.

[53] I. Miers, C. Garman, M. Green, and A. D. Rubin. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *S & P*, 2013.

[54] M. Moser, R. Bohme, and D. Breuker. An inquiry into money laundering tools in the bitcoin ecosystem. In *eCRS*, 2013.

[55] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. http://bitcoin.org/bitcoin.pdf, 2009.

[56] R. Pass and a. shelat. Micropayments for peer-to-peer currencies. Manuscript.

[57] K. Poulsen. Cybercrime supersite 'DarkMarket' was FBI sting, documents confirm. *Wired*, 13 Oct. 2008.

[58] 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.

[59] D. Ron and A. Shamir. How did Dread Pirate Roberts acquire and protect his bitcoin wealth? In *FC*. 2014.

[60] B. Schneier. The vulnerabilities market and the future of security. *Forbes*, May 30, 2012.

[61] S. V. Solms and D. Naccache. On blind signatures and perfect crimes. *Computers Security*, 11(6):581–583, 1992.

[62] M. Stadler, J.-M. Piveteau, and J. Camenisch. Fair blind signatures. In *Eurocrypt*, pages 209–219, 1995.

[63] G. Wood. Ethereum: A secure decentralized transaction ledger. `http://gavwood.com/paper.pdf`, 2014.

[64] A. Young and M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *S & P*, 1996.

A Smart Contract Example

As a warm-up example, Figure 6 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.)

```
Init:      Set $\mathsf{all} := \{\}$, $T_{\text{end}} := 10/12/2015$, $\$\mathsf{price} := 1$.
Register:  On receiving ($\$\mathsf{amt}$, $\mathsf{name}$) 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 6: **Warmup: a simple smart contract for domain name registration.** The formal operational semantics of a contract program is described in Kosba et al. [45].

B Future Directions: Other CSCs

The CSCs we have described in the body of the paper are just a few examples of the broad range of such contracts possible with existing technologies. Also deserving study in a more expansive investigation are CSCs based on emerging or as yet not practical technologies. In this appendix, we give a couple of examples.

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 recently introduced Software Guard eXtension (SGX) set of x86-64 ISA extensions [43], 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.

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 [50]. 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 [36] exists for the sale of 0-days as cyberweaponry [60]. 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 [58].

SGX could enable proofs of validity of a 0-days: app

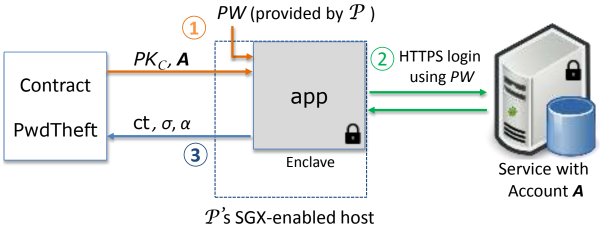

Figure 7: Diagram of execution of $\mathsf{PwdTheft}$ with application app running on SGX-enabled platform. The steps of operation are described in text.

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.

C Preliminaries

Our CSCs rely on a cryptographic building block called non-interactive zero-knowledge proofs (NIZK). We adopt exactly the same for definitions for NIZKs as in Kosba et al. [46]. For completeness, we restate their definitions below.

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

C.1 Non-Interactive Zero-Knowledge Proofs

A non-interactive zero-knowledge 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 $\mathsf{crs} \leftarrow \mathsf{KeyGen}_{\mathsf{nizk}}(1^\lambda, \mathcal{L})$: 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.
- $(\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda, \mathcal{L})$: Generates a simulated common reference string $\widehat{\mathsf{crs}}$, trapdoor τ, and extract key ek
- $\pi \leftarrow \widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \mathsf{stmt})$: Uses trapdoor τ to produce a proof π without needing a witness

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

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

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 for all non-uniform polynomial-time adversary $\mathcal{A}$, we have that

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

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. Otherwise, if $(\mathsf{stmt}, w) \notin \mathcal{L}$, the experiment aborts.

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, we say a NIZK for a language $\mathcal{L}$ is (strongly) simulation sound extractable *iff* there exists an extractor $\mathcal{E}$ such that for all polynomial-time adversary $\mathcal{A}$, the following holds:

$$\Pr\left[\begin{array}{lcl} (\widehat{\mathsf{crs}}, \tau, \mathsf{ek}) \leftarrow \widehat{\mathcal{K}}(1^\lambda) & & (\mathsf{stmt}, \pi) \notin Q \text{ and} \\ (\mathsf{stmt}, \pi) \leftarrow \mathcal{A}^{\widehat{\mathcal{P}}(\widehat{\mathsf{crs}}, \tau, \cdot)}(\widehat{\mathsf{crs}}, \mathsf{ek}) & : & (\mathsf{stmt}, w) \notin \mathbb{R}_{\mathcal{L}} \text{ and} \\ w \leftarrow \mathcal{E}(\widehat{\mathsf{crs}}, \mathsf{ek}, \mathsf{stmt}, \pi) & & \mathcal{V}(\widehat{\mathsf{crs}}, \mathsf{stmt}, \pi) = 1 \end{array}\right]$$
$$= \mathsf{negl}(\lambda)$$

where Q is the set of statement-proof pairs generated by the oracle calls to $\widehat{\mathcal{P}}$.

C.2 Succinct Non-Interactive ARguments of Knowledge (SNARKs)

A SNARK is a NIZK scheme that is perfectly complete, computationally zero-knowledge, and with the additional properties of being succinct and having a knowledge extractor (which is a stronger property than soundness):

Succinctness. A SNARK is said to be succinct if an honestly generated proof has $\mathsf{poly}(\lambda)$ bits and that the verification algorithm $\mathcal{V}(\mathsf{crs}, \mathsf{stmt}, \pi)$ runs in $\mathsf{poly}(\lambda) \cdot O(|\mathsf{stmt}|)$ time.

Knowledge extraction. Knowledge extraction property says that if a proof generated by an adversary is accepted by the verifier, then the adversary "knows" a witness for the given instance. Formally, a SNARK for language $\mathcal{L}$ satisfies the knowledge extraction property *iff:*

For all polynomial-time adversary $\mathcal{A}$, there exists a polynomial-time extractor $\mathcal{E}$, such that for all uniform advice string z,

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

Note that the knowledge extraction property implies computationally soundness (defined for NIZK), as a valid witness is extracted.

C.3 Instantiating Simulation Sound Extractable NIZKs

The *composability* of cryptographic building blocks such as zero-knowledge proofs is of vital importance when constructing larger protocols. In practice, this ensures that each cryptographic building block or protocol does not interfere with other (possibly concurrently executing) protocol instances. It has been shown [42] that simulation sound extractability for NIZKs is roughly equivalent to universal composable [26, 27, 29] security for NIZKs.

In our implementations, we use the techniques described by Kosba et al. [46] to realize simulation sound extractable NIZKs (formally defined in Section C.1) from regular SNARKs (formally defined in Appendix C.2).

```
Ideal-NaiveKeyTheft

Init:    Set state := INIT.
Create:  Upon recipient of ("create", $\$\mathsf{reward}, \mathsf{pk}_{\mathcal{V}}, T_{\mathrm{end}}$) from
         some contractor $\mathcal{C}$:
           Notify ("create", $\$\mathsf{reward}, \mathsf{pk}_{\mathcal{V}}, T_{\mathrm{end}}, \mathcal{C}$) to $\mathcal{S}$.
           Assert $\mathsf{ledger}[\mathcal{C}] \geq \$\mathsf{reward}$.
           $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] - \$\mathsf{reward}$
           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 $\mathsf{match}(\mathsf{pk}_{\mathcal{V}}, \mathsf{sk}_{\mathcal{V}}) = 1$
           Notify ("claim", $\mathcal{P}$) to $\mathcal{S}$.
           If $\mathcal{C}$ is corrupted, send $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{S}$.
           $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$
           Send $\mathsf{sk}_{\mathcal{V}}$ to $\mathcal{C}$
           Set state := CLAIMED.
           /* reward goes to 1st successful claim*/
Timer:   If state = CREATED and current time $T > T_{\mathrm{end}}$:
           Set $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$
           Set state := ABORTED.
```

Figure 8: 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.

D Formal Protocols for Key Theft Contract

D.1 Ideal Program for the Naive Key Theft

The ideal program for the naive key theft contract is given in Figure 8. We stress that here, this naive key theft ideal program is different from the strawman example in the main body (Figure 3). For ease of understanding, Figure 3 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 D.4

```
Contract-NaiveKeyTheft

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

Create: Upon receiving (“create”, $reward, $\mathsf{pk}_\mathcal{V}$, $T_{\mathrm{end}}$)
        from some contractor $\mathcal{C} := (\mathsf{pk}_\mathcal{C}, \ldots)$:
            Assert state = INIT.
            Assert $\mathsf{ledger}[\mathcal{C}] \geq$ $reward.
            $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] -$ $reward.
            Set state := CREATED.

Intent: Upon receiving (“intent”, cm) from some pur-
        ported perpetrator $\mathcal{P}$:
            Assert state = CREATED.
            Assert that $\mathcal{P}$ did not send “intent” earlier.
            Store cm, $\mathcal{P}$.

Claim: Upon receiving (“claim”, ct, $\pi$, $s$) from $\mathcal{P}$:
            Assert state = CREATED.
            Assert $\mathcal{P}$ sent (“intent”, cm) earlier such that
            $\mathsf{cm} := \mathsf{comm}(\mathsf{ct}||\pi, s)$.
            Assert that $\pi$ is a valid NIZK proof (under crs)
            for the following statement:
                $\exists r, \mathsf{sk}_\mathcal{V}$ s.t. $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_\mathcal{C}, (\mathsf{sk}_\mathcal{V}, \mathcal{P}), r)$
                  and $\mathsf{match}(\mathsf{pk}_\mathcal{V}, \mathsf{sk}_\mathcal{V}) = \mathtt{true}$

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

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

Figure 9: A naïve, flawed key theft contract (lacking incentive compatibility). The notation $\mathsf{pk}_\mathcal{C}$ serves as a short-hand for $\mathcal{C}$.epk. This figure is a repeat of Figure 3 for the readers' convenience.

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.

```
Prot-NaiveKeyTheft

Contractor $\mathcal{C}$:
  Create: Upon receiving input (“create”, $reward, $\mathsf{pk}_\mathcal{V}$, $T_{\mathrm{end}}$,
          $\mathcal{C}$):
              Send (“create”, $reward, $\mathsf{pk}_\mathcal{V}, T_{\mathrm{end}}$) 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 $\pi$ for the following state-
              ment:
                  $\exists r, \mathsf{sk}_\mathcal{V}$ s.t. $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}_\mathcal{C}, (\mathsf{sk}_\mathcal{V}, \mathcal{P}), r)$
                    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, $\pi$, $s$) to
              $\mathcal{G}$(Contract-NaiveKeyTheft).
```

Figure 10: User-side programs for naive key theft. The notation $\mathsf{pk}_\mathcal{C}$ serves as a short-hand for $\mathcal{C}$.epk.

D.2 Full Protocol for Naive Key Theft

The contract and full protocols for naive key theft are given in Figures 9 and 10. Specifically, Figure 9 is a repeat of Figure 3 for the readers' convenience.

Theorem 1 *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).

D.3 Proofs for Naive Key Theft Contract

We now prove Theorem 1. 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.

D.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 $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$ 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 $\mathcal{S}(\mathsf{simP})$. 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 $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$. simP forwards the message to the simulated inner contract functionality $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$, 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 $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$ 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 $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$ 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 $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$ 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 $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$.

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 $\mathcal{G}(\mathsf{Contract\text{-}NaiveKeyTheft})$ 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 $\mathcal{S}(\mathsf{simP})$ 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 $\mathcal{F}(\mathsf{Ideal\text{-}NaiveKeyTheft})$.

- Simulator simP receives a message (“create”, \$reward, $\mathsf{pk}_\mathcal{V}, T_{\text{end}}, \mathcal{C}$): do nothing.

D.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 $\mathcal{G}$(Contract-NaiveKeyTheft) functionality. Since all messages to the $\mathcal{G}$(Contract-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 $\mathcal{E}$.*

D.4 Extension to Incentive Compatible Key Theft Contract

Ideal program. The ideal program for an incentive compatible key theft contract is given in Figure 11.

Contract. The incentive compatible key theft contract is given in Figure 12 (a repeat of Figure 4 for the readers' convenience).

Protocol. The user-side programs for the incentive compatible key theft contract are supplied in Figure 13.

Theorem 2 (Incentive compatible key theft contract) *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, then the protocol described in Figures 12 and 13 securely emulates* $\mathcal{F}$(Ideal-NaiveKeyTheft).

Proof: A trivial extension of the proof of Theorem 1, the naive key theft case.

```
Ideal-KeyTheft

Init: Set state := INIT.
Create: Upon recipient of ("create", $\$\mathsf{reward}, \mathsf{pk}_{\mathcal{V}}, T_{\mathrm{end}}, \Delta T$) from some contractor $\mathcal{C}$:
    Same as Ideal-NaiveKeyTheft (Figure 8), and additionally store $\Delta T$.
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", $\Pi_{\mathrm{revoke}}$) from some $\mathcal{R}$:
    Notify $\mathcal{S}$ of ("revoke", $\Pi_{\mathrm{revoke}}$)
    Assert $\Pi_{\mathrm{revoke}}$ 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 $\Delta T$ time elapsed since successful **Claim**:
    $\mathsf{ledger}[\mathcal{P}] := \mathsf{ledger}[\mathcal{P}] + \$\mathsf{reward}$;
    Set state := ABORTED.
  Else if current time $T > T_{\mathrm{end}}$ and state $\neq$ ABORTED:
    $\mathsf{ledger}[\mathcal{C}] := \mathsf{ledger}[\mathcal{C}] + \$\mathsf{reward}$.
    Set state := ABORTED.
```

Figure 11: Thwarting revoke-and-claim attacks in the key theft ideal program.

```
Contract-KeyTheft

Init: Set state := INIT. Let $\mathsf{crs} := \mathsf{KeyGen}_{\mathrm{nizk}}(1^{\lambda})$ denote a hard-coded NIZK common reference string generated during a trusted setup process.
Create: Same as in Contract-NaiveKeyTheft (Figure 9), except that an additional parameter $\Delta T$ 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", $\Pi_{\mathrm{revoke}}$) from some $\mathcal{R}$:
    Assert $\Pi_{\mathrm{revoke}}$ 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 $\Delta T$ 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 $T > T_{\mathrm{end}}$ 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 12: Key compromise CSC that thwarts revoke-and-claim attacks. Although supercially written in a slightly different manner, this figure is essentially equivalent to Figure 4 in the main body. We repeat it here and write the contract with respect to the differences from Figure 9 for the readers' convenience.

Prot-KeyTheft

Contractor $\mathcal{C}$:

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

Send ("create", \$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 10), 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 13: User-side programs for incentive compatible key theft.

E Formal Protocols for Public Document Leakage

E.1 Formal Description

Ideal program for public document leakage. We formally describe the ideal program for public document leakage in Figure 14.

Contract. The contract program for public leakage is formally described in Figure 15, which is a repeat of Figure 2 for the readers' convenience.

Protocol. The protocols for public leakage are formally described in Figure 16.

Theorem 3 (Public leakage) *Assume that the encryption scheme* (Enc, Dec) *is perfectly correct and semantically secure, the NIZK scheme is perfectly complete and computationally zero-knowledge, then the protocol described in Figures 2 and 16 securely emulates* $\mathcal{F}$(Ideal-PublicLeaks).

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

E.2 Proofs for Public Document Leakage

E.2.1 Ideal World Simulator

The wrapper part of $\mathcal{S}$(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 $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}_{\text{enc}}(1^{\lambda})$. The simulator gives $(\mathsf{crs}, \mathsf{pk})$ to the environment $\mathcal{E}$, and remembers sk.

The simulator $\mathcal{S}$(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 $\mathcal{F}$(Ideal-PublicLeaks) – and this message is routed through $\mathcal{S}$. simP now generates an msk using the honest algorithm. For $i \in [n]$, pick $\mathsf{ct}_i \overset{\$}{\leftarrow} \{0,1\}^{\ell}$ 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 $\mathcal{G}$(Contract-PublicLeaks). Upon receiving a challenge set Ω from the ideal functionality, use the same Ω for simulating $\mathcal{G}$(Contract-PublicLeaks).

```
Ideal-PublicLeaks

Init: Set state $= \textsc{init}$, and donations $:= \{\}$.
Create: Upon receiving (“create”, $M$, $T_{\text{end}}$) 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 $\Omega$ to the adversary $\mathcal{S}$.
    Set state $:= \textsc{created}$.
Confirm: Upon receiving (“confirm”) from leaker $\mathcal{C}$:
    Assert state $= \textsc{created}$.
    Send $\{m_i\}_{i \in \Omega}$ to the adversary $\mathcal{S}$.
    Set state $:= \textsc{confirmed}$.
Donate: Upon receiving (“donate”, \$amt) from some purchaser $\mathcal{P}$:
    Notify $\mathcal{S}$ of (“donate”, \$amt, $\mathcal{P}$)
    Assert state $= \textsc{confirmed}$
    Assert ledger$[\mathcal{P}] \geq$ \$amt.
    Set ledger$[\mathcal{P}] :=$ ledger$[\mathcal{P}] -$ \$amt.
    donations $:=$ donations $\cup \{(\$\text{amt}, \mathcal{P})\}$.
Accept: Upon receiving (“accept”) from $\mathcal{C}$:
    Notify (“accept”, $\mathcal{C}$) to the ideal adversary $\mathcal{S}$.
    Assert state $= \textsc{confirmed}$.
    ledger$[\mathcal{P}] :=$ ledger$[\mathcal{P}] +$ sum(donations)
    Send $M$ to the ideal adversary $\mathcal{S}$.
    Set state $:= \textsc{aborted}$.
Timer: If state $= \textsc{confirmed}$ and $T > T_{\text{end}}$: $\forall(\$\text{amt}, \mathcal{P}) \in$ donations: let ledger$[\mathcal{P}] :=$ ledger$[\mathcal{P}] +$ \$amt. Set state $:= \textsc{aborted}$.
```

Figure 14: Ideal program for public leaks.

```
Contract-PublicLeaks

Init: Set state $:= \textsc{init}$, and donations $:= \{\}$. Let crs $:=$ KeyGen$_{\text{nizk}}(1^\lambda)$, pk $:=$ KeyGen$_{\text{enc}}(1^\lambda)$ denote hardcoded public parameters generated through a trusted setup.
Create: Upon receiving (“create”, $c_0$, $\{\text{ct}_i\}_{i=1}^n$, $T_{\text{end}}$) from some leaker $\mathcal{C}$:
    Set state $:= \textsc{created}$.
    Select a random subset $\Omega \subset [n]$ of size $k$, and send (“challenge”, $\Omega$) to $\mathcal{C}$.
Confirm: Upon receiving (“confirm”, $\{(\kappa_i, \pi_i)\}_{i \in \Omega}$) from $\mathcal{C}$:
    Assert state $= \textsc{created}$.
    Assert that $\forall i \in S$: $\pi_i$ is a valid NIZK proof (under crs) for the following statement:
        $\exists(\text{msk}, r_0)$, s.t. $(c_0 = \text{Enc}(\text{pk}, \text{msk}, r_0))$
                $\wedge (\kappa_i = \text{PRF}(\text{msk}, i))$
    Set state $:= \textsc{confirmed}$.
Donate: Upon receiving (“donate”, \$amt) from some purchaser $\mathcal{P}$:
    Assert state $= \textsc{confirmed}$.
    Assert ledger$[\mathcal{P}] \geq$ \$amt.
    Set ledger$[\mathcal{P}] :=$ ledger$[\mathcal{P}] -$ \$amt.
    donations $:=$ donations $\cup \{(\$\text{amt}, \mathcal{P})\}$.
Accept: Upon receiving (“accept”, msk, $r_0$) from $\mathcal{C}$:
    Assert state $= \textsc{confirmed}$
    Assert $c_0 = \text{Enc}(\text{pk}, \text{msk}, r_0)$
    ledger$[\mathcal{C}] :=$ ledger$[\mathcal{C}] +$ sum(donations)
    Send (“leak”, msk) to all parties.
    Set state $:= \textsc{aborted}$.
Timer: If state $= \textsc{confirmed}$ and $T > T_{\text{end}}$: $\forall(\$\text{amt}, \mathcal{P}) \in$ donations: let ledger$[\mathcal{P}] :=$ ledger$[\mathcal{P}] +$ \$amt. Set state $:= \textsc{aborted}$.
```

Figure 15: A contract PublicLeaks that leaks a secret M to the public in exchange for donations. This figure is a repeat of Figure 2 for the readers’ convenience.

```
Prot-PublicLeaks

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

As leaker $\mathcal{C}$:

  Create: Upon receiving ("create", $M := \{m_i\}_{i\in[n]}, T_{\mathrm{end}}, \mathcal{C}$) from the environment $\mathcal{E}$:
      $\mathsf{msk} \overset{\$}{\leftarrow} \{0,1\}^\lambda$
      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 $r_0 \overset{\$}{\leftarrow} \{0,1\}^\lambda$ and compute $c_0 := \mathsf{Enc}(\mathsf{pk}, \mathsf{msk}, r_0)$.
      Send ("create", $c_0$, $\{\mathsf{ct}_i\}_{i\in[n]}$, $T_{\mathrm{end}}$). to $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.

  Challenge: Upon receiving ("challenge", $\Omega$) from $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$:
      For $i \in \Omega$: compute a NIZK proof $\pi_i$ for the statement using witness $(\mathsf{msk}, r_0)$:
          $\exists(\mathsf{msk}, r_0)$, 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 $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.

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

As purchaser $\mathcal{P}$:

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

  Leak: Upon receiving ("leak", $\mathsf{msk}$) from $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$:
      Download $\{(i, \mathsf{ct}_i)_{i\in[n]}\}$ from $\mathcal{G}(\mathsf{Contract\text{-}PublicLeaks})$.
      For $i \in [n]$, output $\mathsf{Dec}(H(\mathsf{PRF}(\mathsf{msk}, i)), \mathsf{ct}_i)$.
```

Figure 16: User-side programs for public leaks.

- *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", \$amt, $\mathcal{P}$) to an honest donor, simulator simP receives ("donate", \$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", \$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.

[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.

- *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 $\mathcal{E}$.

Remark. Our overall proof structure for this variant is the same as that 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.

F Supplemental Details for Document Leakage

F.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.

F.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 17 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 17 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.

F.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:

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

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 [31] 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 $(\mathsf{sk}_{\mathsf{vrf}}, \mathsf{pk}_{\mathsf{vrf}})$ and an associated pseudorandom function F. It takes as input a value i and outputs a pair (σ, π), where $\sigma = F_{\mathsf{sk}_{\mathsf{vrf}}}(i)$, and π is a NIZK proof of correctness of σ. The NIZK π can be verified using $\mathsf{pk}_{\mathsf{vrf}}$.

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 [31] 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 $F_{\mathsf{sk}_{\mathsf{vrf}}}(i) = H(i)^x$ 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.

G Calling-Card Crimes

In this appendix, we explain how to construct CSCs for crimes beyond the website defacement achieved by SiteDeface.

In $\mathsf{SiteDeface}$, 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 \$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 \$deposit $> p \times$ \$reward, adversaries are economically disincentivized from brute-force guessing of calling cards. Commission-fairness then relies on economic rationality.

Finally, we note that it is also possible to *implement anti-CSCs using calling cards*. For example, an anonymous reward could be made available for returning a stolen painting, informing on a criminal, etc.

H Formal Definition for Calling-Card Criminal Contracts

We formally describe the ideal program for Calling-Card Criminal Contracts in Figure 18. We make the simplifying assumption that the trusted data feed DataFeed emits pre-processed calling-card data that are directly checked by the program. It should also be noted that the *Params* argument denotes a general list of attributes that are adapted to the context. For example, in the context of the SiteDeface CSC discussed earlier (Figure 5), *Params* will include the service public key, the webpage URL, and the desired statement.

```
                         Ideal-CallingCard

  Init:  Set state := INIT.
Create:  Upon receiving ("create", $reward, Params, DataFeed, $T_{\text{end}}$) from some contractor $\mathcal{C}$:
           Notify ("create", $reward, Params, DataFeed, $T_{\text{end}}, \mathcal{C}$) to $\mathcal{S}$.
         Assert ledger[$\mathcal{C}$] $\geq$ $reward.
         ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] $-$ $reward
         Set state := CREATED.
Commit:  Upon receiving ("commit", cc) from some perpetrator $\mathcal{P}$:
           Assert state = CREATED.
           Notify ("commit", $\mathcal{P}$) to $\mathcal{S}$.
           Assert cc was not sent before by any other perpetrator.
           Assert this is the first commit received from $\mathcal{P}$.
           Store ($\mathcal{P}$, cc).

Reward:  Upon receiving ("reward", Params', cc') from DataFeed:
           Assert state $\neq$ ABORTED.
           Notify ("reward", Params', cc, DataFeed) to $\mathcal{S}$.
           Assert Params' = Params
           Find the Perpetrator $\mathcal{P}$ who sent a ("commit", cc) such that cc = cc'.
           If $\mathcal{P} \neq$ nil
             Set ledger[$\mathcal{P}$] := ledger[$\mathcal{P}$] + $reward
           else
             Set ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] + $reward
           Set state := ABORTED.
 Timer:  If state = CREATED and current time $T > T_{\text{end}}$:
           Set ledger[$\mathcal{C}$] := ledger[$\mathcal{C}$] + $reward
           Set state := ABORTED.
```

Figure 18: Ideal program for a generalized calling card CSC.

```
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 17: Public leakage contract implemented on top of Ethereum.