The Sting Framework: Proving the Existence of Superclass Adversaries

Mahimna Kelkar
Cornell Tech

Yunqi Li
UIUC

Nerla Jean-Louis
UIUC

Carolina Ortega Pérez
Cornell Tech

Kushal Babel
Cornell Tech

Andrew Miller
UIUC

Ari Juels
Cornell Tech

Abstract

We introduce *superclass accountability*, a new notion of accountability for security protocols. Classical notions of accountability typically aim to *identify specific adversarial players* whose violation of adversarial assumptions has caused a security failure. Superclass accountability describes a different goal: to *prove the existence of adversaries capable of violating security assumptions*.

We develop a protocol design approach for realizing superclass accountability called the *sting framework* (SF). Unlike classical accountability, SF can be used for a broad range of applications *without making protocol modifications* and even when *security failures aren't attributable* to particular players.

SF generates proofs of existence for superclass adversaries that are *publicly verifiable*, making SF a promising springboard for reporting by whistleblowers, high-trust bug-bounty programs, and so forth.

We describe how to use SF to prove the existence of adversaries capable of breaching the *confidentiality* of practical applications that include Tor, block-building infrastructure in web3, ad auctions, and private contact discovery—as well as the *integrity* of fair-transaction-ordering systems. We report on two end-to-end SF systems we have constructed—for Tor and block-building—and on experiments with those systems.

1 Introduction

Imagine that an intelligence agency has developed a powerful new internal service called TorBreaker. TorBreaker *violates the network security assumptions that underpin Tor*. It traces Tor traffic and identifies the websites a target client is visiting. In other words, TorBreaker breaks Tor's privacy.

A whistleblower inside the intelligence agency has black-box access to TorBreaker. She doesn't know whether or not TorBreaker has been used yet to violate users' privacy. Either way, she wants to *prove to the world the existence of an adversary that can break Tor*. What can she do?

This hypothetical example illustrates the focus of our work.

In general, security proofs for cryptographic or security protocols rely on a set of assumptions about the capabilities or behavior of an adversary. In this work, we advance a broad framework for *proving violation of these assumptions*.

In multi-party computation protocols [5], for example, a standard assumption is that an adversary corrupts less than t of n players. Protocols making use of trusted execution environments (TEEs) rely on (typically strong) assumptions about the isolation of the TEE from adversarial processes [60]. Protocols enforcing network-layer privacy, such as Tor, assume limited adversarial control of the network [40].

What happens, however, when these assumptions are broken? Most protocol designs have *all-or-nothing* security. This means that violation of a modeling assumption then results in catastrophic failure: A complete loss of confidentiality or execution integrity. Moreover, such failures are often *silent*—that is, unobservable by protocol participants. This is especially the case with confidentiality violations, which can leak information with no visible effect on a system's functionality.

Several approaches have been proposed to mitigate catastrophic protocol failures caused by inaccurate adversarial modeling [35, 36]. An especially popular one is *accountability* [26, 49]. Broadly speaking, a security protocol is accountable if, when the protocol fails because of misbehavior from players, it is possible to determine (or even prove the identities of) at least some misbehaving players.

Traitor-tracing (e.g., [8, 9, 19]) is an extensively studied form of accountability used in practice [37] to identify players that collude to create pirate decryption capabilities. Accountability can also, however, involve identifying corrupted players in BFT consensus [20, 56, 67], demonstrating subverted setup of cryptographic protocols [1], identifying unauthorized access or handling of data [24, 31, 48], and so forth.

Traitor-tracing schemes in particular have seen some adoption in practice. But they have proven fragile in the wild [37] (and often only allow accountability / tracing by *a trusted entity* with a secret tracing key). Beyond traitor tracing, accountable protocols have seen little uptake in practice.

Current approaches to accountability have two notable

drawbacks. First, they require *purpose-designed protocols*, i.e., protocols that include accountability mechanisms by design. For example, traitor-tracing systems assign different sets of decryption keys to different players. Thus accountability mechanisms cannot easily be added to existing systems. Second, nearly all accountability protocols specifically *assume that failures can be attributed to corrupted players*. They are inapplicable in the common case where security failures are instead the result of faulty protocols or bugs and *there may not exist identifiable corrupted players*.

Our goal in this work is to introduce an alternative framework for accountability that addresses the limitations of prior schemes and is more broadly and practically deployable.

1.1 Superclass Accountability

We introduce a new, broader notion of accountability that we call *superclass accountability*. Our goal is *not* to blame corrupted players or instances of misbehavior by corrupted players, as in standard notions of accountability [49]. Instead, our goal is *system-level* accountability—meaning proof that a system's security model has been violated, in the sense that an adversary exists with capabilities outside the model. We refer to such an adversary as a *superclass adversary*, meaning that it is more powerful than defined by system's security model.

Suppose a protocol π is designed for security against adversaries in a class $\mathbb{A}$. Achieving superclass accountability with respect to π and some superclass $\mathbb{A}^* \supset \mathbb{A}$ then means the following. If the adversary $\mathcal{A}$ happens to be within the class $\mathbb{A}^* \setminus \mathbb{A}$ instead, then it is possible to generate a proof of this fact that will convince a third party verifier Judge. The proof does not need to specify the capabilities of $\mathcal{A}$, nor which players are involved in realizing $\mathcal{A}$.

A superclass adversary $\mathcal{A}$ could be one that has corrupted players more aggressively than expected (e.g., corrupted $t+1$ players when security assumes at most t players). But it could equally well be that $\mathcal{A}$ can exploit a system or protocol bug, e.g., can break the isolation of a TEE through a side-channel attack—and may not correspond to any particular player.

Superclass accountability is *weaker* in one sense than standard accountability: It does not aim to blame corrupted players. But it is at the same time *stronger*, in that it can capture a broader range of system failures, and can moreover work directly with existing systems. It is consequently more amenable to practical application than standard notions of accountability, as we show in this work.

1.2 Sting Framework (SF)

To realize superclass accountability for a range of protocols, we introduce an approach that we call the *Sting Framework* (SF). As an approach to proving the existence of superclass adversaries, SF has several key properties: (1) SF is applicable to a *wide range of protocols*; (2) SF can in many cases generate *publicly verifiable proofs*; and (3) SF requires *no modification to existing protocols*.

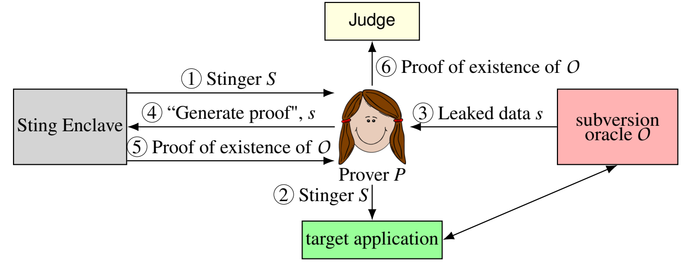

Figure 1: Basic sting protocol flow for proving a superclass adversary's ability to violate target-application confidentiality. A prover $\mathcal{P}$ interacts with the Sting Enclave to obtain a stinger S. $\mathcal{P}$ then sends it to a target application, and uses a subversion oracle O to retrieve a secret s derived from S (such as decrypting S). Importantly, the secret s should only be known to the target application and the Sting Enclave. Therefore, when the user sends s back to the Sting Enclave, if s is correct, it would prove to the Sting Enclave that there exists some subversion oracle O. The Sting Enclave can then generate an attestation that O exists, which $\mathcal{P}$ can send to other parties.

SF assumes that a superclass adversary $\mathcal{A}$ is realizable through an interface to abuse a target protocol, called a *subversion oracle* O. A subversion oracle could represent any of a number of ways that a system is exploited: It could be a paid dark-market service, a private service stood up by an adversary, or a functionality created by a white-hat hacker to prove the existence of an exploit. SF may be applied to a given O *independently of how* O *is realized by adversaries*.

Figure 1 depicts the general flow for proving the existence of a superclass adversary that violates a confidentiality property. (SF can also be applied to attacks on protocol integrity.)

In SF for confidentiality, a user / prover $\mathcal{P}$ who has access to O seeks to generate a proof of the existence of O—and consequently of a superclass adversary $\mathcal{A}$. To do so, $\mathcal{P}$ relies on a trusted third party (TTP). For the applications we propose in this paper, this trusted third party will be a TEE—we will refer to it as the Sting Enclave. However, the TTP could also be a standalone trusted entity or an MPC committee.

The key idea in SF is for $\mathcal{P}$ to use the TTP to generate a specially crafted protocol input S called a "stinger"[1]. S contains encrypted data s to which no one should have access given the existence only of "expected" adversaries $\mathcal{A} \in \mathbb{A}$. $\mathcal{P}$ then uses O to gain access to the secret s, which $\mathcal{P}$ then uses to prove the existence of a superclass adversary $\mathcal{A} \notin \mathbb{A}$.

A simplified application example helps illustrate the idea.

Example 1 (SF for private contact-discovery)**.** Consider a private contact-discovery service `Contact`. It enables users to discover contacts in their contact lists that are also fellow users of a messaging app such as Signal [55]. The security

[1]The term "sting" in SF refers to the idea of a "sting operation".

model for `Contact` assumes that adversaries *cannot directly read private contact lists*. Call this class of adversaries $\mathbb{A}$.

Suppose, however, that `Contact` has a critical vulnerability and someone has created a dark-market subversion oracle $\mathcal{O}$ that leaks users' contact lists. Thus there exists a superclass adversary $\mathcal{A} \notin \mathbb{A}$ that *can directly learn private contact lists*.

A user / prover $\mathcal{P}$ with access to $\mathcal{O}$ can prove $\mathcal{O}$'s existence as follows (with a number of omitted details). $\mathcal{P}$ obtains a stinger S from a Sting Enclave consisting of a secret contact list s that is input—in encrypted form—into `Contact`. $\mathcal{P}$ calls $\mathcal{O}$ to learn s. $\mathcal{P}$ then reveals s to the Sting Enclave—proving her ability to extract s.

An attestation from the Sting Enclave that $\mathcal{P}$ can learn s is itself a proof of existence of a superclass adversary $\mathcal{A} \notin \mathbb{A}$ because the existence of $\mathcal{O}$ implies such an $\mathcal{A}$. (More precisely, $\mathcal{A} \in \mathbb{A}^* \setminus \mathbb{A}$ for an adversarial superclass $\mathbb{A}^*$ that depends on details of the SF protocol.)

Observe that *the standard notion of accountability is not applicable here*. There may be no specific player(s) executing the protocol for `Contact` that can be blamed for the existence of $\mathcal{O}$. Moreover, if adversaries make only clandestine use of $\mathcal{O}$, there may be *no visible evidence that* `Contact` *is broken*—and thus no way for honest users outside dark markets even to know that an accountability protocol should be invoked.

1.3 Target Applications

In this work, we show how SF can provide superclass accountability—*with no application-level modification*—for several popular applications. We implement and experimentally validate SF systems for two of them:

- *Tor:* We introduce an SF system for proving the existence of an adversary that can break Tor, in the sense of identifying the website visited by a targeted Tor client. An example application of our SF system would be publicly verifiable whistleblowing in the case of, e.g., an intelligence agency attacking Tor successfully [65].
- *Block-building:* Construction of blocks in Ethereum is increasingly performed by entities, known as *builders*, who must enforce confidentiality on input transactions (to prevent arbitrage attacks). TEEs are a recently adopted approach to securing builder infrastructure. We show SF enables proof of confidentiality failures in TEE-based builder infrastructure, e.g., [7]. The ability to prove the existence of such leakages will build trust in the builder market, which is necessary for its decentralization [84].

For other applications where experiments would be less informative, we outline SF-system constructions. These include private contact discovery, and private ad auctions.

While our focus in this paper is on confidentiality, to show that SF also applies to adversaries breaking other forms of security, we also discuss how SF can be applied to show violations of *integrity properties* on blockchain transaction ordering [14, 15, 44, 45], and malicious behavior in auctions.

SF proofs for the applications we consider are *publicly verifiable*. In addition to ensuring strong transparency, public verifiability can support trustworthy bug-bounty programs—an important goal given frequent reports of bug bounties going unpaid [83]. Public verifiability can in principle support even automated bug-bounty smart contracts on blockchains [11].

1.4 Contributions

Our summary contributions in this work are:

- *Superclass accountability:* We introduce and define a new notion of accountability, *superclass accountability*, with broader application than traditional notions (Section 2).
- *Sting Framework (SF):* We introduce a framework for superclass accountability (Section 2) that: (1) Applies to a *wide range of protocols*, (2) Generates *publicly verifiable proofs*, and (3) Requires *no modification to existing protocols*.
- *Practical applicability of SF:* We demonstrate the broad range of uses of SF by discussing its use to prove confidentiality breaches (Sections 3 and 4) and integrity breaches (Section 5) across a range of practical applications, including Tor, block-building infrastructure, ad auctions, and fair transaction-ordering systems.
- *End-to-end implementations:* We implement and experimentally evaluate SF systems for Tor and block-building infrastructure. We provide an application agnostic interface for easier development of Sting protocols. (Section 6).

2 Sting Formalism

Basic setup. Let κ denote the system security parameter. We adopt the standard Interactive Turing Machine (ITM) approach to model protocol execution [16]. Informally, each party is modeled as an ITM, and a protocol details how parties interact with each other. Execution (including message-sending between parties) is handled by a special environment machine $\mathcal{Z}(1^\kappa)$. An adversary $\mathcal{A}$ interacts with $\mathcal{Z}$ to receive information about and control specific parts of the execution, according to pre-determined constraints. While nodes are often individually categorized into "honest" or "adversarial" (with honest nodes following the protocol faithfully and adversarial nodes being under the control of $\mathcal{A}$), we refrain from following this convention in the case of sting protocols in order to allow for broader classes of corruption (for example, the corruption of only a specific sub-routine but for all nodes).

An ideal functionality abstracts out the required goals of the system in an "ideal" world, while a protocol details the real-world interaction between parties. We consider functionalities between a set of n parties. Inputs to the parties (often from clients) are modeled as provided by $\mathcal{Z}$. We use $\mathsf{EXEC}^\pi(\mathcal{A}, \mathcal{Z}, \kappa)$ to denote the random variable for the execution of protocol π with environment $\mathcal{Z}$, and adversary $\mathcal{A}$; each view in its support completely defines an execution. $\mathsf{OUT}_U(\mathsf{view})$ denotes the output of party U in view. Security

is now defined as follows: A protocol π is said to emulate ideal functionality $\mathcal{F}$ w.r.t. $\mathbb{A}$ if for all $\mathcal{A} \in \mathbb{A}$, there exists an efficient simulated adversary $\mathcal{S}$ such that for all $\mathcal{Z}$, the ensembles $\mathsf{EXEC}^{\pi}(\mathcal{A}, \mathcal{Z}, \kappa)$ and $\mathsf{EXEC}^{\mathcal{F}}(\mathcal{S}, \mathcal{Z}, \kappa)$ are indistinguishable. For environment $\mathcal{Z}$ and ITM U, let $\mathcal{Z}^U$ denote the new environment which incorporates U within $\mathcal{Z}$ although for simplicity, we may still refer to U as a separate party.

We now formally define sting protocols below. Recall that we are interested in proving the existence of a "superclass" adversary—i.e., one that belongs to a more powerful class than the one originally considered for the protocol at hand.

Definition 1 (Superclass Accountability / Sting Protocol)**.** Consider adversarial classes $\mathbb{A}$ and $\mathbb{A}^*$ with $\mathbb{A} \subset \mathbb{A}^*$. Consider an ideal functionality $\mathcal{F}$ and a protocol π which emulates $\mathcal{F}$ w.r.t. $\mathbb{A}$. A sting protocol for $(\mathcal{F}, \pi, \mathbb{A}, \mathbb{A}^*)$ is a tuple $(\mathcal{P}, \mathsf{Judge})$ where $\mathcal{P}$ is an ITM (we call $\mathcal{P}$ the *prover*) and Judge is a deterministic algorithm satisfying the following for all sufficiently large κ:

- (Completeness) For all $(\mathcal{A}^*, \mathcal{Z})$ where $\mathcal{A}^* \in \mathbb{A}^* \setminus \mathbb{A}$, except with $\mathsf{negl}(\kappa)$ probability over $\mathsf{view} \leftarrow\$ \mathsf{EXEC}^{\pi}(\mathcal{A}^*, \mathcal{Z}^{\mathcal{P}}, \kappa)$, it holds that $\mathsf{Judge}(1^{\kappa}, \mathsf{OUT}_{\mathcal{P}}(\mathsf{view})) \Rightarrow 1$.
- (Soundness / No Framing) For all ITM $\mathcal{P}^*$ and all $(\mathcal{A}, \mathcal{Z})$ where $\mathcal{A} \in \mathbb{A}$, except with negligible probability over $\mathsf{view} \leftarrow\$ \mathsf{EXEC}^{\pi}(\mathcal{A}, \mathcal{Z}^{\mathcal{P}^*}, \kappa)$, it holds that $\mathsf{Judge}(1^{\kappa}, \mathsf{OUT}_{\mathcal{P}^*}(\mathsf{view})) \Rightarrow 0$.

We can relax these to ε-completeness and ε-soundness, i.e., the properties hold except with $\varepsilon = \varepsilon(\kappa)$ probability. We allow for any ε satisfying $\varepsilon(\kappa) < 1/2 - \kappa^{-c}$ for a constant $c > 0$.

Setting for confidentiality applications. Looking ahead, we will be particularly interested in the following setting: the superclass adversary is able to learn some leakage s, which cannot be learned by an adversary in $\mathbb{A}$. We model this leakage through a *subversion oracle* O which is accessible to the superclass adversary—a very general model that does not require specification of *how* the superclass adversary actively learns the secret. Concretely, we model $\mathbb{A}^*$ as the class $\mathbb{A}$ *but also with access to* O; we write $\mathbb{A}^* = \mathbb{A}^O$.

The goal now for a sting protocol is essentially to prove existence of O. The protocol will be carried out by any $\mathcal{P}$ who has access to O. Notably, in many practical examples, the prover may actually be a party corrupted by the superclass adversary—*a whistleblower* of sorts. As an example, the party may only have access to O due to being part of a collusion. Here, we need to assume that the corrupted node can still spawn the code for $\mathcal{P}$ as a sub-routine not corrupted by $\mathcal{A}$. This models a realistic scenario where even a node operator corrupted by the adversary can run a *different* machine to perform the sting protocol without alerting the adversary.

2.1 Formalism Implications

Adversarial detection of sting protocols. It is natural to think that a sting protocol's execution must not be detectable by $\mathcal{A}^* \in \mathbb{A}^* \setminus \mathbb{A}$ in order to be performed successfully. Indeed if it can be detected whether a sting protocol has been initiated by $\mathcal{P}$, then it can modify its responses to prevent the sting protocol from succeeding. As an example, for our confidentiality setting, the subversion oracle O may refuse to respond when it detects an ongoing sting protocol.

Even so, notice that the sting protocol could be run repeatedly (as independent instances) until it is not detected and a valid sting proof is obtained. In essence, a sting protocol will still work even if it is sometimes detectable, but this may affect its efficiency.

Undetectability is partially captured within the completeness notion—if $\mathcal{A}^*$ is able to detect a sting proof attempt by $\mathcal{P}$ with probability ε, and e.g., stops the protocol, then $\mathcal{P}$'s sting protocol will succeed with probability at most $1 - \varepsilon$, i.e., it cannot be $(\varepsilon' < \varepsilon)$-complete. At the same time, false positives should also be minimized—in that light, we define undetectability below as the adversarial advantage in the game for detecting sting attempts.

Definition 2 (ε-Undetectability)**.** Let $(\mathcal{P}, \mathsf{Judge})$ be a sting protocol for $(\mathcal{F}, \pi, \mathbb{A}, \mathbb{A}^*)$. We say that $(\mathcal{P}, \mathsf{Judge})$ is ε-undetectable if for any $\mathcal{A}^* \in \mathbb{A}^* \setminus \mathbb{A}$, no PPT distinguisher can distinguish between the ensembles $\mathsf{EXEC}^{\pi}(\mathcal{A}^*, \mathcal{Z}, \kappa)$ and $\mathsf{EXEC}^{\pi}(\mathcal{A}^*, \mathcal{Z}^{\mathcal{P}}, \kappa)$ with probability more than ε.

We provide more details on undetectability, with a particular focus on our setting, in Section 3.2.1.

Reducing completeness error. Running multiple protocol instances also enables the amplification or boosting of a sting protocol with ε-completeness error to one with negligible error. In general, if $(\mathcal{P}, \mathsf{Judge})$ has completeness error ε, then $(\mathcal{P}_{\ell}, \mathsf{Judge})$, where $\mathcal{P}_{\ell}$ runs $\mathcal{P}$ internally ℓ times (can be parallelized), will have completeness error ε^{ℓ}. There does however exist a practical trade-off between the efficacy of a sting protocol, and the time (and often cost) it takes to generate a valid proof—undetectability is concretely relevant for this as we discuss in our applications in Sections 3.

Reducing soundness error. Generic soundness amplification is trickier since it requires some notion of sequentiality to be enforced among the rounds. Fortunately, for our sting protocols in particular, we can amplify soundness by enforcing sequentiality in a non-black-box way, through the Sting Enclave. We provide details in Section 3.2 and Appendix A.

3 Sting Protocols for Privacy Leakage

We consider two broad categories for sting protocols: ones for proving privacy leaks (this section and Section 4), and ones for proving integrity violations (Section 5). Our sting protocols differ substantially for these two settings.

This section provides a general structure of sting protocols for proving leakage of private secrets. We later detail practical examples—on Tor (Section 4.1), block-building in blockchains like Ethereum (Section 4.2), and Signal contact

discovery (Section 4.3). We also provide implementations for the first two, and analyze them experimentally in Section 6.

3.1 Motivation and General Framework

Consider a protocol π among n nodes where a secret s is kept hidden from individual nodes (or more generally from adversary $\mathcal{A}$ from a presumed class $\mathbb{A}$). What if $\mathcal{A}$ happens to be in a more powerful class, i.e., a *superclass* $\mathbb{A}^*$, and can therefore learn s? Our goal in this section is to investigate *general* techniques to *publicly* prove the existence of such a superclass adversary that can learn s (or more generally $\mathcal{L}(s)$ for some leakage function $\mathcal{L}$). A core requirment for us is to be able to work within existing systems—i.e., *without requiring any protocol modifications*. Note that these proofs will not require that s was actually leaked—although it may have been—only that such an $\mathcal{A}$ exists.

An important observation that ultimately motivates our approach is that there are settings where it is possible to show the presence of a leakage, but impossible to identify exactly which parties were responsible for it. Consider the following:

Example 2 (Secret Sharing)**.** In the (t,n)-secret-sharing setting, n parties hold *shares* of a secret s such that any t of them can reconstruct the secret, but any malicious coalition smaller than t will learn no information about it. In typical constructions (such as a commonly-used one from Shamir [66]), a t-sized coalition can recover not only the secret, but also the shares of other parties. Even if the secret s is leaked, it is not possible to identify responsible parties, i.e., individual accountability is not feasible. Still, we would like to show here that e.g., "something has gone wrong."

More broadly, a natural question is how to equip a whistleblower—who knows of a subversion scheme to leak secrets—to prove this fact. While new cryptographic primitives could be explored to offer accountability for individual parties, the settings for these are typically limited, and furthermore, it is challenging to retrofit them to existing systems, both for efficiency and complexity reasons.

While individual accountability can often be infeasible, on the flip side, simply showing knowledge of a leaked secret s (through e.g., *proofs-of-knowledge* (PoKs)) can be insufficient for many real-world applications, and especially the ones we consider. In some settings, vanilla PoKs do indeed trivially provide superclass accountability—as an example, some cryptographic protocols use a trusted *setup* phase that generates a persistent secret s which, if leaked, breaks the protocol; here, proving knowledge of s effectively shows a malicious setup.

The *sting* setting. But, as we shall see, in many systems, we want to protect against s being leaked within the protocol, even though it is presumed that s is already known to some party—typically the end user. Our focus is on these settings. Take, for instance, our Signal use-case from Example 1. Here, s is the contact list of the user $\mathcal{U}$—who of course knows it because she created it. This means that a simple PoK will not suffice to demonstrate existence of a superclass adversary, since (a malicious) $\mathcal{U}$ could on her own act as a prover $\mathcal{P}$ to complete the PoK for s, even if there is no superclass adversary.

Proving superclass adversaries here requires *interaction* from the prover $\mathcal{P}$ to *inject* specially crafted inputs to the system—hence the term *sting*, signaling the sting operation carried out by $\mathcal{P}$ to detect malicious behavior.

Detecting *active exploitation* of secrets. Suppose that a superclass adversary $\mathcal{A} \in \mathbb{A}^*$ has the power to learn a secret s. But what if $\mathcal{A}$ either does not reconstruct s or does not make further use of it? Standard security definitions still consider such an $\mathcal{A}$ to have broken privacy and "won" the security game—a conservative, yet reasonable worst-case way to define adversarial success. In real-world scenarios however, adversaries of concern are typically ones that take further action. $\mathcal{A}$ might use s itself or even e.g., sell generalized access to secrets in a dark marketplace—i.e., *actively leak secrets*.

Sting protocols hinge critically on this distinction between an adversary that has broken protocol privacy in the standard sense and one that actively uses or leaks secrets. In the latter cases, sting protocols will allow *any party* $\mathcal{P}$ with the ability to learn s—for instance, knowledge from the aforementioned dark marketplace—to *whistleblow* or prove this publicly.

In fact, this $\mathcal{P}$ could even be a sub-entity within $\mathcal{A}$—for instance, one party within a set of colluding protocol nodes. In practice, $\mathcal{A}$ is seldom the monolithic entity typical in cryptographic modeling, and instead, is likely comprised of distinct entities, with potentially competing interests. Viewed another way, sting protocols can facilitate defection within a collusion.

Modeling leakage through a *subversion* oracle. We model active adversarial leakage of secrets through a generic *subversion* oracle functionality $\mathcal{O}$. Abstractly, this allows us to generically model leakage without requiring to know the actual cause—this models not only corruption of players, but also to other factors, such as implementation level bugs or side-channel attacks. For now, suppose that querying $\mathcal{O}$ fully leaks the secret s to the caller. We will consider more general types of leakage later (see Section 3.3).

For the rest of this section, we consider a specific superclass $\mathbb{A}^*$ which is similar to $\mathbb{A}$ except that now, the adversary also has access to $\mathcal{O}$—and can therefore learn s. In effect, the sting protocols will be carried out by any $\mathcal{P}$ who also has access to this $\mathcal{O}$ in order to publicly prove its existence. Section 4 illustrates how even this simple model for a superclass adversary captures a wide range of applications.

3.2 Protocol Design

Basic structure. Our sting protocols for showing privacy leakages employ an instantiation of an ideal trusted party—in the examples we explore, through a trusted hardware enclave such as Intel SGX [21]. We refer to this functionality used by

the prover as the *Sting Enclave*, or simply *enclave* where it's clear from context.

The outline of our basic sting protocol is conceptually simple. At a high level, the Sting Enclave is first used to generate a *stinger* S using a random secret input s, which will then be used by $\mathcal{P}$ within the *target* protocol or application π, for which we aim to construct a sting protocol.

The *stinger* S hides s so that it will be unknown to anyone without access to subversion $\mathcal{O}$—this provides soundness against a malicious prover. $\mathcal{P}$ will now use its access to $\mathcal{O}$ to retreive s, and show this to the enclave, which verifies correctness. Later in this section, we elaborate on how to handle a general leakage $\mathcal{L}(s)$ provided by $\mathcal{O}$. Now, upon verification of s, the enclave can further generate a signature of this fact, which can be checked publicly and without requiring private state—including by a smart contract—through the enclave's *remote attestation* functionality [21, 75]. We use Judge to denote this public verification process.

While the above outline is simple, as we will see, there are several nuances that come into play when concretely designing sting protocols.

Communicating with target applications. Cryptographic protocols typically assume secure communication channels between participants, although this is often abstracted out from the model. Our sting protocols rely on such a secure channel—in fact, constructing the stinger will explicitly depend on the details of the channel.

To illustrate why, notice an apparent conflict between the requirements for stingers. While the stinger S needs to hide the secret input s (from the prover), it still needs to maintain the input format expected by the application (otherwise a superclass adversary could easily detect the sting attempt). Intuitively, we get around this by having the Sting Enclave control the secure channel (instead of $\mathcal{P}$), and "injecting" the secret input directly into the channel—this will be our stinger.

As a concrete example, consider communication in practice, which is typically encrypted with TLS. In this setting, we will have the Sting Enclave control the TLS session key. The enclave will now encrypt s (with the appropriate TLS cipher) to obtain the stinger S, which can be given to $\mathcal{P}$ without revealing s. Effectively, in this communication, $\mathcal{P}$ will simply serve as a *forwarder* for the enclave's stinger.

Why the enclave? Notice that a standalone verifier can itself generate the stinger, without use of the enclave—the sting protocol can be thought of as an interactive proof between a prover $\mathcal{P}$ and verifier $\mathcal{V}$, where $\mathcal{P}$ is tasked with proving information leakage from the protocol π (modeled here through $\mathcal{O}$) given a stinger from $\mathcal{V}$. Such a design however, only convinces $\mathcal{V}$, and not any public party. Using an enclave allows us to prove leakage—including partial—publicly and practically. Furthermore, deploying a Judge smart contract as a public verifier enables automated disbursement of bug bounties.

There is a separate concern however—since $\mathcal{P}$ should not know s (without $\mathcal{O}$), it would need to sample S without knowing s. In most cases, this should not be possible with a single prover entity without e.g., a trusted third-party. As an example, when using a secure channel which uses a semantically secure encryption scheme, it should not be feasible to generate a valid stinger. While, a separate committee could emulate the role of the generating the stinger, this committee itself would need to be e.g., majority trusted without the possibility of a superclass adversary within its own ranks.

Nevertheless, we find an interesting insight in the case that the target application also uses a TEE—we call this the *double-TEE* setting (see App. B.1).

3.2.1 Constructing Stingers

We now describe a framework for constructing stingers. Our general finding is that in practice, the process intrinsically depends on the target application. Still, we seek to capture its essence in this section.

Basic model. Let χ denote the distribution of regular user inputs (which may be unknown). For the sting protocol, the enclave will instead sample s from an estimated distribution χ^*. Ideally, χ^* should sufficiently approximate χ so as to make the sting attempt undetectable to the application—this affects the number of rounds required for a valid proof. Choosing χ^* appropriately will be specific to the target application.

The stinger S will be a (typically keyed and randomized) function $F(s)$ of the secret input s—recall that in our context, S will be e.g., the TLS encryption of s, with the session key being held inside the Sting Enclave. More generally, we can define soundness of the stinger S as follows:

Definition 3 (Stinger soundness)**.** The stinger generation is ε-sound if for any (adversarial) prover $\mathcal{P}^*$, the following holds:

$$\mathsf{Adv}^{\chi^*,F}_{\mathsf{snd}}(\mathcal{P}^*) = \Pr[(\mathcal{P}^*(S) \to s') = s \mid s \leftarrow_\$ \chi^*; S \leftarrow_\$ F(s)] < \varepsilon.$$

Let $\mathsf{Adv}^{F}_{\mathsf{inv}}(\mathcal{P}^*)$ denote the same as above except that s is sampled uniformly at random from the domain of χ^*—this is essentially the advantage of inverting a random input to F. Note that for our sting protocols, since we use an encrypted channel to communicate s to the target application, F cannot be inverted except with negligible probability.

Lemma 1. *$\mathsf{Adv}^{\chi^*,F}_{\mathsf{snd}}(\mathcal{P}^*) \leq \mathsf{Adv}^{F}_{\mathsf{inv}}(\mathcal{P}^*) + tvd(\chi^*, U_{|\chi^*|})$ where tvd denotes the total variation distance and $U_{|\chi^*|}$ is the uniform distribution over the domain of χ^*.*

Note that since the distance is with the uniform distribution, the tvd term here can also be bounded by 2^{-h} where $h = H^{\chi^*}_{\mathsf{min}}$ denotes the min-entropy of χ^*.

Undetectability. In addition to soundness against an adversarial prover, we want to ensure that the input generated for the sting protocol appears to come from the true user distribution. In other words, this would make the sting attempt undetectable to the target application. In our model, observe that

undetectability is nothing but the advantage for distinguishing between χ and χ'—this can be bounded by $tvd(\chi,\chi^*)$.

In general, using a good estimate χ^* for the user distribution χ is a highly application-dependent and sometimes challenging problem. As an illustration, in our sting protocol for Signal contact discovery, we were unable to generate fully functional phone numbers inside the SGX enclave—a limitation that fundamentally influenced our protocol design.

Happily, sting protocols can be constructed even for ε-undetectability with large (constant) ε—although at the cost of potentially high sting-protocol round complexity.

Theorem 2 (Number of rounds)**.** *Consider a sting protocol with χ,χ^* as above, and assume that the stinger channel F was perfectly non-invertible. If we require ε_c-completeness and ε_s-soundness, then running n rounds is sufficient with*

$$n > \frac{-12\ln(\varepsilon)\cdot\max\{1-p,1-r\}}{(1-p-r)^2}$$

where $\varepsilon = \max\{\varepsilon_c,\varepsilon_s\}$, $p = tvd(\chi,\chi^*)$, $r = tvd(\chi^*,U_{|\chi^*|})$, and $(1-p-r) > 0$ holds.

In a simpler setting where O provides no output instead of an incorrect one when a sting attempt is detected, we can take $p = 0$.

Sources of entropy. It is easy to see that a sting protocol is inherently limited by the entropy of χ. This χ however, models the *entire* user input string, including e.g., metadata, signatures etc., which can provide additional entropy to what comes solely from the raw input. For instance, in our blockbuilding application (Section 4.2), we use the signature on the sting transaction as a source of additional entropy.

Effectively tying together multiple rounds. When using multiple rounds for soundness amplification, it is imperative to ensure that the prover submits the outputs of all rounds—otherwise, e.g., a malicious prover could run more rounds and only submit the output of the ones it succeeds in (see App. A). Soundness amplification is simple to do in our protocol design by enforcing sequentiality through the Sting Enclave. In particular, the Sting Enclave generates the stinger for a round after the prover submits the output for the previous round.

3.3 Extensions

3.3.1 Partial Leakage

We now consider a generalization where the leakage given by the subversion O is only partial. We look at two complementary axes which can be easily combined: (1) O outputs only with probability ρ; (2) O outputs a randomized leakage $\mathcal{L}(s)$.

(1) If O provides an output only with probability ρ, then by using ℓ rounds, it is easy to see that we get a sting protocol with completeness error $(1-\rho)^\ell$. This works perfectly if the only requirement is to prove existence of a superclass adversary that has access to the leakage—in fact, the Sting Enclave only needs to verify the output of a single round. However, it can affect soundness if we aim to explicitly show existence of a superclass adversary *with the particular probability* ρ *of leakage*. We see this in our Tor application (Section 4.1), where superclass adversaries with differing powers can be considered depending on what fraction of the network they control. Here, a malicious prover could attempt to show existence of a stronger superclass adversary than is the case.

To handle this, notice that the success for each round follows a $\mathsf{Ber}(\rho)$ distribution when O outputs with probability ρ. In this abstraction, a malicious prover is effectively trying to convince the Sting Enclave of distribution $\mathsf{Ber}(\rho')$ (with $\rho' > \rho$) instead of the actual distribution $\mathsf{Ber}(\rho)$. The solution is effectively a distribution testing problem—by running n rounds of the protocol (and essentially obtaining n samples for a distribution $\mathsf{Ber}(\lambda)$ for unknown parameter λ), the Sting Enclave will test whether $\lambda = \rho$ or not. This takes $O(1/\delta^2)$ samples when the acceptable range is $|\rho-\lambda| < \delta$.

(2) Now consider the case that O outputs a (possibly randomized) leakage $\mathcal{L}(s)$ and let D and R denote its domain and range. Intuitively, here, the sting prover claims to have to access to $\mathcal{L}(s)$; it is now up to the Sting Enclave to determine whether or not this is true. This turns out to depend on how easy it is to simulate the output of $\mathcal{L}$ without access to s.

Define $\mathcal{L}$ to be δ-unsimulatable if there exists a PPT distinguisher $\mathcal{D}$, which given s and with at most 1 query, can distinguish, $\mathcal{L}(s)$ from the output of any PPT adversary $\mathcal{B}$ (without access to s) with advantage at least δ. In other words,

$$\left|\Pr[\mathcal{D}^{\mathcal{L}(s)}(s) \Rightarrow 1 \mid s \leftarrow\!\!\$\, D] - \Pr[\mathcal{D}^{\mathcal{B}}(s) \Rightarrow 1 \mid s \leftarrow\!\!\$\, D]\right| \geq \delta.$$

Notice that the distinguisher $\mathcal{D}$ is tasked with the following: Given a description of $\mathcal{L}$, it needs to distinguish whether an input value provided to it came from $\mathcal{L}(s)$ or from a different (sufficiently far away) distribution. This is exactly the identity testing problem studied extensively in the property testing literature [17]. The exact number of samples (or rounds in our case) needed depends on the skew of the distribution with the worst case being roughly $O(\sqrt{|D|})$.

In the sting setting, $\mathcal{P}$ wishes to prove that the leakage it has access to (using O) has distribution $\mathcal{L}(s)$. The idea is that the Sting Enclave runs the distinguisher $\mathcal{D}$ to ascertain whether the leakage shown comes from the claimed distribution.

When $\mathcal{L}$ is δ-unsimulatable, then one run of the sting protocol has soundness error $1-\delta$. Therefore, running the protocol ℓ times will get soundness error $(1-\delta)^\ell$. Concretely, if we require soundness error ε, and we have an $\mathcal{L}$ that is δ-unsimulatable, then we need $\ell = \log(\varepsilon)/\log(1-\delta)$ rounds when $\delta \neq 1$ and 1 round when $\delta = 1$.

3.3.2 Early Leakage

Consider now, a setting where the application guarantees privacy only temporarily—that is, all parties will eventually learn the secret s in a normal execution of the system. This is a common design in practice—for example, an auction where bids

are kept secret until all bids are received, or a blockchain where transactions are only revealed after ordering [14, 27]. Here, a prover needs to show the Sting Enclave that the leakage it obtained was not part of the eventual public reveal.

This is simple to do in a case where the secret is always revealed after time t—the sting proof now needs to be completed within this time. Note that this is practically realizable since SGX enclaves are equipped with local timers.

However, in some settings—most notably our block-building application—the secret s may be publicly revealed at an unknown time. In this case, the previous protocol general structure may no longer work. Instead, we will take the following intuitive approach: the sting prover will use its knowledge of the leakage to perform a publicly verifiable action (on, e.g., a blockchain) which can at any later point establish the *causality between* the prover's leakage and the eventual public release. We give concrete details in Section 4.2.

4 Confidentiality Applications

4.1 The Tor Network

Background. Our first application is for Tor [25], which enables users to communicate with websites anonymously by routing their requests through a randomly chosen path or *circuit* of intermediate relay nodes (run by volunteers around the world) in the network. Tor's design ensures that no relay simultaneously knows the original source and the final destination of the request—each relay is only aware of its immediate neighbors in the communication path. Theoretically, as long as one circuit node is honest and not colluding with others, the privacy of the user's website access is maintained. Typically, at initialization, a 3-length circuit (i.e., 3 intermediate nodes) is chosen to balance anonymity and latency.

However, Tor's privacy can be compromised if a superclass adversary controls a significant number of network relays or observes the traffic between them, potentially linking the data's source to its destination. In this section, we show a sting protocol for demonstrating the existence of such an adversary.

Sting protocol design. To begin, we consider a superclass adversary that can fully deanonymize any user's network destination—the leakage here is captured using the oracle $\mathcal{O}$.

Our sting protocol follows the general structure described earlier. First, the Sting Enclave samples a random destination website (which remains hidden from the prover), and generates a private Tor circuit (the "stinger" circuit) to visit the website. For this, the Sting Enclave runs a standard Tor client within it; we will work with the default 3-hop circuit.

An important note here is that the initial hop (i.e., the first relay node) is already observable by the prover $\mathcal{P}$ by simply observing the enclave's outgoing traffic. As a result, the secret s—which $\mathcal{P}$ wishes to demonstrate access to—corresponds to the latter two hops of the stinger circuit. $\mathcal{P}$, using its access to $\mathcal{O}$, now retrieves the stinger circuit; for simplicity, we assume that $\mathcal{O}$ enables access to the entire circuit (partial leakage can be handled as discussed in Section 3.1). This is then shown to the Sting Enclave, which provides a signature (for public verification by Judge) after checking correctness.

Analysis and discussion. Observe that the efficacy depends on the *undetectability* of the sting protocol, i.e., how easy it is for the adversary to detect an ongoing sting attempt. We make the sting attempt difficult to detect as follows: (1) first, by running a standard Tor client within the Sting Enclave, we mirror the execution trace of a regular user; (2) second, we can sample the destination website from a typical distribution for websites accessed through Tor (c.f., [54] which suggests a power-law distribution). Note that whenever the sting attempt is not detected, for the above $\mathcal{O}$, the protocol always succeeds (i.e., completeness holds with probability 1).

For soundness, we need to prevent a malicious prover from convincing the Sting Enclave of a sting proof without having access to $\mathcal{O}$. If relays were sampled uniformly at random to form the circuit, the probability of of randomly guessing the circuit correctly is $1/\binom{n}{2}$ where n is total number of Tor relays (since the prover will always know the first "entry" node). However, in practice, circuit selection is based on the bandwidth of relays—those with higher bandwidth are more likely to be chosen. Based on experimental analysis [34, 63], in practice, the probability of correctly guessing the circuit may in fact be closer to 1/300 or 1/500. Still, the protocol can easily be repeated multiple times to achieve the required soundness level. Note that we need to ensure sequential execution between attempts—this is easy to do by having the Sting enclave generate new stinger circuits only after a response is received from the prover for the previous execution.

Even if the prover itself controls Tor nodes and can deanonymize the sting circuit using this power, this is no different than it itself being a superclass adversary (or having access to $\mathcal{O}$)—in either case, the sting proof will be valid.

Proving weaker superclass adversaries. We could also consider "weaker" superclass adversaries that only control a fraction α of e.g., the total relay bandwidth and can therefore deanonymize only some circuits. To show the presence of such an adversary, we will again need multiple stinger circuits as we discuss below—the approach follows our discussion on *partial leakage* from Section 3.1.

A concern here is the soundness error—a malicious prover could attempt to show existence of a more powerful superclass adversary (i.e., one that has corrupted a larger fraction of Tor bandwidth) than is the case. This can be done by the malicious prover restarting the Sting Enclave, and requesting a new stinger until it receives one that it can deanonymize. To get around this issue, the Sting Enclave will generate several stinger circuits sequentially, and ask for deanonymization by the prover. Recall that this is effectively distribution testing for $\mathsf{Ber}(p = \alpha^2)$. If the true distribution is $\mathsf{Ber}(\beta^2)$ (i.e., the superclass adversary controls β fraction of the network), and

the claimed distribution is $\mathsf{Ber}(\alpha^2)$, this can be distinguished using $O(1/\delta^2)$ stinger circuits (where $\delta = |\alpha^2 - \beta^2|$).

4.2 Block-Building in Blockchains

Background. In blockchains, transactions sent by users get added to the blockchain ledger by *block proposers* (e.g., Bitcoin miners, or Ethereum validators). Proposers have absolute authority on deciding the transactions and their order within a block; being profit-maximizing, they aim to construct the block most profitable to them. This involves not just transaction fees, but more complex strategies captured under the umbrella term *maximal extractable value* (MEV) [4, 23, 62] that exploit the proposer's ordering authority.

The rise of decentralized finance, however, and the resultant added complexity of crafting the most profitable block has caused proposers to "sell" their ordering power by means of a new division of labor [52, 82]—*searchers* who use specialized and often proprietary strategies to create profitable bundles of user transactions, *builders* who aggregate multiple bundles and compete to create the most profitable block, and proposers, whose only task now is to choose the best block and propagate it within the blockchain consensus protocol. Effectively, through searcher and builder competition, proposers can now profit from their ordering power without having to specialize in the complex strategies themselves.

Searchers leverage their exclusive access to specific user transactions, as well as proprietary algorithms, to construct profitable bundles. The profitability of searchers is crucially dependent on the confidentiality of the contents of their bundles during the builder aggregation phase. In particular, a malicious builder who peeks at the contents of a bundle can steal the exclusive user transactions contained within, manipulate their order, or worse, exploit even the searcher's transactions! To enhance privacy guarantees provided to searchers, many prominent builders have adopted secure environments for bundle aggregation. For example, Flashbots (Ethereum address: 0xc83dad...0965D6) utilize Intel SGX for this purpose.

Privacy concerns for the SGX builder. In addition to concerns about Intel SGX vulnerabilities, the complexity inherent in the builder's software and its seamless integration with other system components also presents significant risks. In fact, a recent incident highlights the risk magnitude: a software glitch resulted in the leakage of bundle transactions, which were then exploited, leading to a loss of \$20 million [47]. Similar vulnerabilities arising from discrepancies between theoretical designs and actual implementations have been identified in the literature [39, 43].

While we remain neutral on the ethical implications of searchers—who may exploit ordinary users—being themselves exploited, it is undeniable that the searcher-builder interaction has become a critical, load-bearing component of the Ethereum ecosystem. We further note that even for builders that do not use SGX, our sting protocol enables proving evidence of malicious behavior by the "trusted" builder.

Sting protocol design. The starting point is to observe that a searcher's transactions do not remain confidential forever—they become public effectively as soon as they are sent to the proposer. As alluded to in Section 3.3, the timed nature of the setting creates a new obstacle especially because the public reveal can happen at an indeterminate time—even the use of a local timer within the Sting SGX Enclave does not suffice.

Instead, we employ a different approach—one that leverages the public nature of the blockchain. Very roughly, the prover will commit on-chain to its (early) knowledge of (the secret) s, at most by the block that s is finalized in. Subsequently, the Sting Enclave, can now use this public blockchain state, to verify early leakage of s.

For simplicity, suppose that the oracle O leaks every bundle (and all transactions within) received by the SGX-builder; partial leakage can be appropriately handled as in Section 3.

Now, the sting protocol works as follows: First, the prover uses the Sting Enclave to create a *stinger* bundle S that encrypts a private transaction s; the bundle is sent encrypted directly to the SGX-builder so that s is not revealed to the prover. The Sting Enclave runs a standard, publicly known searcher algorithm to generate the bundle transactions so as to mimic the behavior of an ordinary searcher; this makes sure that the sting bundle cannot be distinguished by a potentially corrupted SGX-builder. Moreover, note that even if sampled transactions have low entropy (e.g., to look real), note that the transaction signature will provide enough entropy to prevent a malicious prover from attempting a false proof by guessing s (further, as standard, if the entropy is not sufficient for the desired level of soundness, the protocol can be repeated).

Now, the prover uses its access to O to retrieve s, and intuitively will create a new transaction t which contains a *commitment* to its knowledge of s. This is done as follows: Let $C = \mathsf{Com}(s;r)$ denote a cryptographic commitment to s using randomness r. $r' = \mathcal{H}(C)$ will now be used as the randomness to sign transaction t, where the hash function $\mathcal{H}$ is modeled as a random oracle.

Next, the prover goes through the standard searcher-builder auction marketplace, and places the new transaction t in the same or earlier block as s by paying a *competitive bid*. As soon as both s and t are published on the blockchain, the prover can open the commitment (by revealing r) to prove knowledge of s to the Sting Enclave—since t was published no later than s on the blockchain, this is sufficient to show that the prover knew s *prior* to its public release. Concretely, the prover will provide as part of the proof, r, t, and a finalized chain containing t before or in the same block as s. The Sting Enclave now checks the correctness as above, and issues a signature which can be publicly checked by Judge.

We defer some subtleties and analysis to Appendix C.1.

4.3 Signal Contact Discovery

We now describe the sting protocol for private contact discovery, as alluded to in Section 1.2.

Background. Signal is a messaging app well-known for its prioritization of user privacy. In addition to providing end-to-end encryption for user messages, Signal aims to provide additional privacy by also hiding user *metadata*. One example of this is its *contact discovery* protocol [55], which enables a user to learn which of her contacts are also Signal users, without the Signal servers learning her contact list.

To accomplish this, Signal makes use of an Intel SGX enclave. The contact discovery protocol is intuitively as follows: the user sends her contact list to the enclave, which computes its intersection with the set of all Signal users, and returns the result to the user. Communication between the user and the enclave is encrypted so that even the Signal servers cannot eavesdrop on it, maintaining privacy of the user's contact list.

Sting protocol design. We consider the presence of a subversion oracle O that leaks the user's entire contact list allowing the superclass adversary to learn the user's contacts (as before, partial leakage can be handled as in Section 3.1). As an added obstacle, the subversion oracle is careful to not leak the contacts list for a user it deems to be a sting prover.

A strawman sting protocol following our general structure is now as follows: A prover $\mathcal{P}$ can create a new contact list inside the Sting SGX Enclave, so that the list is even hidden from $\mathcal{P}$. The Sting Enclave registers a fresh user with Signal with this contact list; now, if there is a subversion oracle O that leaks the contact list, this can be shown to the Sting Enclave which verifies the leaked contacts to complete the sting proof.

Practical constraints. We ran into some practical issues with this design: First, what makes a contact list look "real"? Arguably, if the Sting Enclave generated a list with random phone numbers, this could be easily distinguished from a real user's contact list, thereby subverting the sting attempt. A second challenge is that this design requires the new user's phone number to be "owned" by the SGX. We were unable to create a fully functioning phone number inside an enclave that was capable of performing the out-of-band authentication (OOBA) required by Signal for registration and spam-prevention.

To solve the first problem, we ask $\mathcal{P}$ to submit a contact list L to the Sting Enclave, which samples a portion of it for Signal's registration protocol. Our approach is for $\mathcal{P}$ to assemble L by merging some of its own contacts with additional *degree-two* contacts, i.e., contacts of its contacts (with their permission). The goal is to simulate the new Signal account being a new member of $\mathcal{P}$'s social circle. This would not be the case if, for instance, the Sting Enclave generated random phone numbers on its own to form L. Instead, we can leverage a vast existing body of work on social-graphs [29, 51, 64, 77]. Although we leave the exact process as future work, from L, the Sting Enclave sub-samples L' to create the new user's contact list. L' serves as the secret s for our sting protocol.

For the second problem, we let $\mathcal{P}$ control the phone number N used for Signal registration by the Sting Enclave, allowing $\mathcal{P}$ to perform Signal's OOBA. Note that this still does not allow $\mathcal{P}$ to learn L'. A concern remains that a malicious prover could attempt to re-register the phone number on a different device, and recover L' using a backup. To prevent this, we require the prover to enable the Signal functionality that creates a PIN for re-registration— this PIN will be chosen by the enclave. We defer further details to App. C.2.

5 Sting Protocols for Integrity

We now illustrate how our sting framework captures a more general framework beyond privacy leakages by showing sting protocols for a number of *integrity* property violations.

General setting and intuition. Consider once again, a protocol π being run by a set of nodes. Suppose that π takes input x and generates an output y that is publicly visible. For now, although not important for our protocols, suppose that π has a single input and output. Here, an integrity property $\mathbb{P}$ on the output is a function of the form $\mathbb{P}(x,y) \to \{0,1\}$ which outputs whether the property was satisfied. A common goal now, is to ensure that $\mathbb{P}$ is satisfied.

This is trivial when $\mathbb{P}$ can be simulated without knowledge of x. An example is when the only requirement on the output is that it needs to be signed by a specified party—here, any verifier can check $\mathbb{P}$ just using the public output y.

But often, it cannot be determined using y, whether or not $\mathbb{P}$ was satisfied. A protocol π designed to guarantee $\mathbb{P}$ typically makes additional assumptions on the participants (e.g., some form of honesty). Here, if the assumption is broken, y may no longer satisfy $\mathbb{P}$, and there may be no way for a public observer to recognize this. This illustrates a *silent* failure.

We aim to demonstrate sting protocols in this domain—specifically, by injecting *stinger* inputs, we can prove violations of $\mathbb{P}$. As we show, this allows detection of a number of failures—even in centralized settings with a dishonest server.

5.1 Transaction Ordering Guarantees

Consider a system where users submit transactions, which are then output as a public sequence according to the system's logic. This models a typical blockchain setting which acts a sequencer for user transactions into a linearly ordered ledger. Such a design can even model a centralized server (e.g., a database) that processes user transactions in a sequence.

In the blockchain context, transaction *ordering*, i.e., the order in which transactions are sequenced and executed by the system, has been shown to be of high importance; adversarial influence can cause exploitation of ordinary users as well as overall system instability [23, 62]. As a potential solution, a number of works [14, 15, 44, 45] (among many others) have proposed protocols that seek to reduce adversarial

influence by guaranteeing certain integrity properties on the output transaction sequence. Still, all these protocols assume for instance, honest majority among protocol participants. Furthermore, simply looking at the output, one cannot determine whether or not the ordering property was satisfied. In other words, a violation of the protocol's honesty assumption can lead to a silent failure of the ordering guarantee.

We describe sting protocols for proving such violations.

Simplified setting. We model a centralized server which processes transactions by first sequencing them. The server is typically assumed to be honest here—our goal therefore is to detect when the server misbehaves, i.e., when the output does not conform to the ordering rule. As an example, suppose that we want to enforce first-come-first-serve (FCFS) ordering; we note that our protocol can be analogously transformed for other ordering rules (e.g., [53]). In Appendix C.3, we provide details on how to extend our protocol to a distributed setting.

Protocol details. The core idea is essentially to embed trapdoor information within a transaction, which can later be revealed to demonstrate server misbehavior. Consider the following strawman: an informer submits a transaction A to the server, and embeds a cryptographic commitment of the receipt on this transaction from the server into a second transaction B (in e.g., the randomness in the signature submitted along with the transaction data). Now, if B gets ordered before A by the server, the informer has convincing proof of misbehavior—all it needs to do is to reveal the commitment. We note that even if the server does not provide an explicit receipt for transactions, an implicit receipt can be interpreted at the TLS network layer. For this, the informer protocol will need to be run inside an SGX enclave to ensure that the TLS transcript is not forged.

This sting protocol is always sound, but may not always catch server violations, especially if the server manipulates the ordering only in specific cases. Still, one can imagine that the protocol can be run continuously, or by several users. Even one server violation caught may cause sufficient harm to its reputation of honesty detracting it from behaving dishonestly.

Tampering as a service. Apart from the server itself manipulating the ordering, a larger threat is the server(s) essentially *selling the rights* to this manipulation on an open marketplace—i.e., choosing the transaction ordering given by the highest bidder. This is not a hypothetical scenario—it already happens on blockchains like Ethereum [62,82], and is a major reason that protocols with strong ordering guarantees but requiring honesty assumptions are challenging to deploy.

We make the observation that the existence of such a marketplace would actually improve the efficacy of our sting protocols. For one, the marketplace can function as a subversion leakage oracle—if user transaction details are leaked through it to bidders, then an informer, posing as a bidder could show this leakage similar to our protocols in Section 3.

Surprisingly, even if there is no leakage, such a marketplace actually provides a unique opportunity for the sting protocols to work *by taking part in the manipulation*. For instance, in our previous protocol to show that the ordering between two transactions was incorrect, the informer could directly effect this change by going through the marketplace and buying the rights to reorder transactions. In our modeling, this functions essentially as a "tamper" instruction on the subversion service. This can significantly improve the efficacy of our protocol.

As a general philosophy, sting protocols can help prevent these "tampering as a service" marketplaces. If the seller cannot distinguish whether the buyer is a "legitimate" party interested in the tampering service, or an informer, then it may become unprofitable for such a marketplace to be created.

Sting infrastructure as part of the server design. Remarkably, we can integrate the sting infrastructure within the server itself, and allow the server to make claims about integrity properties, *without* the need for external users to carry out a sting protocol—the idea is to have the server continuously attempt to sting itself through a trusted component such as an enclave within it. We provide details in Appendix C.3.

5.2 Malicious Behavior in Auctions

A second example is for detecting malicious auctioneer behavior in a second-price auction setting. While second-price auctions are truthful—i.e., incentivize bidders to bid their true value, a concern is that the auctioneer can improve its revenue by maliciously posing as a bidder and bidding strategically. This poses a problem even if bids are sealed. In fact, there is evidence of similar behavior in the real world as brought to light by the Justice department's lawsuit against Google for unfairly manipulating its ad auctions [58].

In Appendix C.4, we show a simple example of how a sting prover, aware of such a manipulation, can prove it publicly. The core idea is to use the Sting Enclave to generate a secret bid b, and win the auction in a way that the bidder's payment could only have been influenced by an auctioneer's fake bid.

6 SF Implementation and Evaluation

We design a concrete unified framework to aid in the development of sting protocols for different target applications. For concreteness, we use Intel SGX as the Sting Enclave TEE, and an Ethereum smart contract as Judge—for proof final verification, and release of any bounty. Algorithm 1 describes the general interface. For different applications, the major differences are in the stinger generation, and the required evidence. Our prototype implementations leverage Gramine [76] to run unmodified applications within SGX enclaves.

Sting enclave setup. The SGX Sting Enclave is initialized with two identifiers: `MRENCLAVE`, which is the hash of its contents (code and data), and `MRSIGNER`, which is the identity of its creator. To enable sting verification, the Sting Enclave will need to prove to the remote Judge contract that it runs the

Algorithm 1 SF Implementation Pseudocode (Chronological)

```
1:  ProvisionEnclave(C) → (pk,report,sig_IAS,cert_IAS):       ▷ Sting Enclave
2:      pk_Sting, sk_Sting = EthKeyGen()
3:      report = {MRENCLAVE, MRSIGNER, UserReportData=pk_Sting, ...}
4:      quote = QuotingEnclave(report)
5:      sig_IAS, cert_IAS = IAS(quote, report)
6:      Seal(sk_Sting)
7:  DeployJudgeContract:                                      ▷ initiator
8:      Preapprove a list of MRENCLAVEs
9:      Fund reward pool
10: RegisterEnclave(report, sig_IAS, cert_IAS):               ▷ prover
11:     pk_IAS = VerifyCert(cert_IAS, pk_Intel)
12:     pk_Sting, MRENCLAVE = VerifyAtt(sig_IAS, report)
13:     Require(PreApproved(MRENCLAVE))
14:     Approve(pk_Sting)
15: GenStinger(C) → β:                                        ▷ Sting Enclave
16:     Sample stinger from context C: S = Sample()
17:     Send S to target application
18:     Seal(S)
19:     (optional) Leak backdoor β to prover
20: Leak data L from target application                       ▷ Subversion Oracle
21: MakeEvidence(L, β, C) → E:                                ▷ prover
22:     Locate stinger S in L (using β)
23:     Assign evidence of S leakage to E
24: VerifyEvidence(E) → (prf, sig_Sting):                     ▷ Sting Enclave
25:     S, sk_Sting = Unseal()
26:     if Verify(S, E) then
27:         prf = GenProof()
28:         sig_Sting = Sign(sk_Sting, prf)
29: CollectBounty(sig_Sting, prf, pk_Sting):                  ▷ Judge
30:     Require(Approved(pk_Sting))
31:     if VerifySig(sig_Sting, prf, pk_Sting) then Issue bounty
```

expected sting program within it; this is achieved via SGX's EPID-based remote attestation [41].

For this, the enclave will first need to be *provisioned* before the protocol. In particular, the enclave generates a report of its identity and state, signed by its private key, and submits this to Intel's Quoting Enclave (QE). The QE signs the report with its EPID private key to create a quote verifying the report's authenticity. The quote and report are then verified by Intel's Attestation Service (IAS) using the corresponding EPID public key. Upon validation, IAS provides a signature and a certificate, enabling remote verifiers to confirm the quote's legitimacy. For our concrete implementation, since Judge is an Ethereum smart contract, the Sting Enclave will generate an Ethereum-compatible key pair $(\mathtt{sk}_{\mathtt{Sting}}, \mathtt{pk}_{\mathtt{Sting}})$. The public key $\mathtt{pk}_{\mathtt{Sting}}$ is embedded in the UserReportData field of the report, which allows Judge to verify that the sting proof has been authenticated by the Sting Enclave. The corresponding secret key $\mathtt{sk}_{\mathtt{Sting}}$ is securely sealed by the enclave (e.g., stored in encrypted form on external untrusted storage).

Gramine introduces significant overheads, due to a high-latency enclave initialization process[2], but this needs to be done only once is not time-sensitive in the context of our sting protocols. In particular, the ProvisionEnclave operation takes about 26s, but only 0.285s without Gramine initialization.

After provisioning with IAS, the Sting Enclave will further register itself with the Judge smart contract.

Judge **contract.** An initator, such as an administrator of the target application will deploy the Judge smart contract on Ethereum. The main role of Judge is to handle registration of a Sting Enclave, verify its attestation during the sting proof, and manage the release of a reward for a prover supplying a valid proof. The cost of such a smart-contract based verification is minimal: RegisterEnclave has a gas cost of $246,000$ (about $14), while CollectBounty (which verifies the enclave signature) has a gas cost of $93,000$ (about $5).

Experiment setup. Our benchmarks were conducted on a machine with an Intel Xeon E-2276G CPU and 64GB of RAM, running Ubuntu 20.04. We set up container-based local networks for both applications, ideal for reproducing our results[3]. The machine operates on Linux kernel version 5.15.0-92-generic and uses the SGX driver version 2.11.54c9c4c.

6.1 The Tor Network

We implement our sting protocol for Tor and experimentally benchmark its efficiency. Algorithm 2 (Appendix D) gives the pseudocode; the Tor-specific parts only add about 100 lines of Python code to the Sting Framework interface.

Implementation details. To avoid detection of the sting protocol by the target application, we do the following: (1) The Sting Enclave runs an off-the-shelf [61] Tor client[4]; (2) A public relay list is used to generate a random "sting" circuit and destination address that mirrors an typical user's access.

As a proof of concept, we model a subversion oracle that targets only exit relays (the final circuit hop). Our analysis should therefore provide a conservative bound even for more powerful subversions that may arise as a result of complex attacks discussed in the literature [3, 38, 50].

Once the sting Tor circuit is established, the Tor client (running inside the Sting Enclave) generates a stream through the circuit and directs the exit relay to connect to a specified destination address. The stream ID—unlike the circuit ID, which varies at each relay—is global and visible to the exit relay. This means that an O which models corruption of the exit relay can link stream IDs to their respective destinations. For efficiency reasons, instead of providing the entire output of O to the Sting Enclave, we can do better by enabling the sting prover to identify the correct sting circuit, amongst potentially several circuits leaked by O. For this, the Sting Enclave will deliberately leak the stream ID to the prover.

Evaluation. While the Tor sting protocol is not time-sensitive, it is still useful to assess whether the running time becomes prohibitively long when the adversary controls only

[2]Due to Gramine's unoptimized implementation, SGX provisioning is split into two stages: report generation and quote generation; each stage requires separate enclave initialization, adding to the latency.

[3]https://anonymous.4open.science/r/sting-17C8/README.md

[4]Due to Gramine limitations, a child process initiates a new enclave. As a result, our core SF program and the Tor client operate in separate enclaves.

a fraction α of the relay bandwidth, as this requires multiple sequential executions as discussed in Section 4.1.

For this, we conducted a timing analysis. The setup of starting a local Tor client and sampling a sting circuit takes about 1.35s; using this circuit to connect to the destination website (through WAN) takes about 3s. Verifying that the sting prover found the correct circuit takes minimal time—with a batch of 100 circuits taking less than 0.01s. In turn, each round of the sting protocol will conservatively take 5s.

As one calculation, if 25% of the relay bandwidth is controlled by the superclass adversary, allowing deanonymization of about 7% of circuits [34], the sting protocol will take about 4 hours to achieve a soundness error of 2^{-40}, but only about 25 minutes to reject the null hypothesis (that there is no superclass adversary) with a typical significance level of 0.05.

6.2 Block-Building in Blockchains

Implementation details. We use the Flashbots SGX-builder [27] for Ethereum as the target application. Algorithm 3 provides the pseudocode.To simulate a superclass adversary and model leakage by O, we modify the builder code to leak all received transaction bundles.

Now, within our Sting Enclave, we run a standard public searcher algorithm [28] to generate a "stinger" transaction bundle, in a manner indistinguishable from the activities of ordinary searchers. Instead of using bundle transactions (which have low entropy) as the secret, as alluded to in Section 4.2, we instead embed the secret into the signature of a bundle transaction. More concretely, we use a random r as the random nonce for the ECDSA signature (distinct from the Ethereum transaction nonce) on a transaction `s`. The transaction bundle is sent to the target application (SGX builder) via TLS.

As in the Tor example, for efficiency, the Sting Enclave can leak the hash of `S` to the prover—to help the prover find the correct stinger transaction within all the bundles leaked by O. Once the prover obtains `s`, as described in Section 4.2, it embeds a (Pedersen) commitment to it inside the signature of a second transaction `t`. This `t` is then sent to the blockchain with a high enough gas price to ensure inclusion in the same or earlier block as `s`. The verification by the Sting Enclave is simple—the prover simply provides it a finalized Ethereum block sequence that contains `t` no later than `s`. Soundness is not a concern here—entropy from the signature randomness should already suffice for a negligibly small soundness error.

Latency. Recall that the sting protocol here needs to be time-sensitive. To prove early leakage, the critical sequence (as highlighted in Algorithm 3) pertains to using `s` to craft `t` and getting it included in a block no later `s`. We analyze the latency of this critical sequence. In a local network setting (assuming O responds instantly), the added latency is 2.4s; variation in the size of the sting bundle has negligible effect on this. In a typical WAN setting (i.e., communication with remote entities), the total estimated latency is about 3.3s. This amounts to about 27% of the typical block time of 12 seconds on Ethereum. Note that other parts of the protocol only need to be executed once, and are not time-sensitive. Verification of the sting proof takes about 1.63s (post Gramine initialization).

Cost implications. The gas cost for the transactions `s` and `t` is about 21,000 ($1.25). In other words, the cost would not be prohibitive even if several attempts are required.

7 Related Work

Accountable protocols. Accountability is a recurring theme in the cryptography and security literature; several protocols have been constructed for detecting e.g., maliciously sharing of cryptographic functionalities (through traitor tracing [9,19] and leakage-deterring schemes [46]), safety violations in BFT consensus [57, 67], malicious generation of cryptographic parameters [1,32] etc. Another line of research (see, e.g., [13, 30]) allows identifying players who perform sensitive actions, e.g., identify private data, without necessarily being malicious.

A core feature of accountable systems is identifying malicious parties within the protocol execution. In contrast, sting protocols simply demonstrate the existence of what would otherwise be a silent failure, often without identifying e.g., exactly which component failed or which parties acted maliciously. But, giving up on this feature is precisely what allows sting protocols to be constructed for a broader class of systems. In a sense, the goals are incomparable—sting protocols are both weaker (in strength), and stronger (in generality). This is illustrated by our ability to work with existing systems *without modifications*—in contrast, adding accountability can require a complete rework of the system design, with potential downstream effects on efficiency and legacy-compatibility.

Honeypots / honey objects. Honeypots [68] are fake systems designed to entice, observe, and monitor adversaries. Other honey objects (e.g., [2,6,42]) have a similar goal. While these techniques carry an undetectability requirement like that of sting protocols, and may help detect breaches, they require special-purpose systems and importantly do not construct proofs of adversarial existence.

Software-exploit proofs. A small line of prior works has sought to construct proofs of the existence of code vulnerabilities. These include Cheesecloth [22], which constructs zero-knowledge proofs of knowledge of invariant violations for code running in LLVM, and Reverie [33], which proves exploits in microprocessor code. Unlike sting protocols, these systems assume access to the target code and their use of ZK-proofs scales only to relatively small code instances. Sealed-glass proofs [75] involve the use of TEEs for a similar goal and likewise require prover access to the target code.

8 Conclusion

We have introduced *sting protocols*, a conceptually simple new approach for demonstrating silent failures of security guarantees by proving the existence of *superclass adversaries*—adversaries that are stronger than what is assumed by a protocol's model. Sting protocols are a new twist on accountability: they provide weaker guarantees than existing approaches (by e.g., not identifying corrupted players), but as a result are more general—usable with a broad range of applications and deployable *with no protocol modifications*. We demonstrated these protocols by reporting on two end-to-end prototypes, for Tor and block building, outlining approaches for several other widely used systems.

References

[1] Prabhanjan Ananth, Gilad Asharov, Hila Dahari, and Vipul Goyal. Towards accountability in CRS generation. In *Eurocrypt*, pages 278–308, 2021.

[2] Frederico Araujo, Kevin W Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *CCS*, pages 942–953, 2014.

[3] Daniel Arp, Fabian Yamaguchi, and Konrad Rieck. Torben: A practical side-channel attack for deanonymizing tor communication. In *CCS*, pages 597–602, 2015.

[4] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. Clockwork finance: Automated analysis of economic security in smart contracts. In *IEEE S&P*, pages 2499–2516, 2023.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, 1988.

[6] Maya Bercovitch, Meir Renford, Lior Hasson, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Honeygen: An automated honeytokens generator. In *IEEE ISI*, pages 131–136, 2011.

[7] bert. Post mortem: April 3rd, 2023 mev-boost relay incident and related timing issue. Flashbots Forum, https://collective.flashbots.net/t/post-mortem-april-3rd-2023-mev-boost-relay-incident-and-related-timing-issue/1540, Apr. 2023.

[8] Dan Boneh and Matthew Franklin. An efficient public key traitor tracing scheme. In *CRYPTO*, pages 338–353, 1999.

[9] Dan Boneh, Aditi Partap, and Lior Rotem. Accountability for misbehavior in threshold decryption via threshold traitor tracing. In *CRYPTO*, pages 317–351, 2024.

[10] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

[11] Lorenz Breidenbach, Phil Daian, Florian Tramèr, and Ari Juels. Enter the hydra: Towards principled bug bounties and exploit-resistant smart contracts. In *USENIX Security*, pages 1335–1352, 2018.

[12] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018.

[13] Mike Burmester, Yvo Desmedt, Rebecca N Wright, and Alec Yasinsac. Accountable privacy. In *Security Protocols: 12th International Workshop*, pages 83–95. Springer, 2006.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.

[15] Christian Cachin, Jovana Micic, Nathalie Steinhauer, and Luca Zanolini. Quick order fairness. In *FC*, pages 316–333, 2022.

[16] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–147, 2001.

[17] Clément L. Canonne. A survey on distribution testing: Your data is big. but is it blue. *Theory of Computing Library Graduate Surveys*, (9):1–100, 2020.

[18] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. Controlled data races in enclaves: Attacks and detection. In *USENIX Security*, pages 4069–4086, 2023.

[19] Benny Chor, Amos Fiat, and Moni Naor. Tracing traitors. In *CRYPTO*, pages 257–270, 1994.

[20] Pierre Civit, Seth Gilbert, and Vincent Gramoli. Polygraph: Accountable byzantine agreement. In *ICDCS*, pages 403–413, 2021.

[21] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016.

[22] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. Cheesecloth: Zero-knowledge proofs of real world vulnerabilities. In *USENIX Security*, pages 6525–6540, 2023.

[23] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *IEEE S&P*, pages 585–602, 2020.

[24] Ivan Damgård, Chaya Ganesh, Hamidreza Khoshakhlagh, Claudio Orlandi, and Luisa Siniscalchi. Balancing privacy and accountability in blockchain identity management. In *CT-RSA*, pages 552–576, 2021.

[25] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. Tor: The second-generation onion router. In *USENIX Security*, pages 303–320, 2004.

[26] Joan Feigenbaum, Aaron D Jaggard, and Rebecca N Wright. Towards a formal model of accountability. In *NSPW*, pages 45–56, 2011.

[27] Flashbots. https://github.com/flashbots/geth-sgx-gramine.

[28] Flashbots. https://github.com/flashbots/simple-arbitrage.

[29] Chris Fleizach, Michael Liljenstam, Per Johansson, Geoffrey M. Voelker, and Andras Mehes. Can you infect me now? malware propagation in mobile phone networks. In *WORM*, page 61–68, 2007.

[30] Christina Garman, Matthew Green, and Ian Miers. Accountable privacy for decentralized anonymous payments. In *FC*, pages 81–98, 2017.

[31] Shafi Goldwasser and Sunoo Park. Public accountability vs. secret laws: can they coexist? a cryptographic proposal. In *WPES*, pages 99–110, 2017.

[32] Vipul Goyal. Reducing trust in the pkg in identity based cryptosystems. In *CRYPTO*, pages 430–447, 2007.

[33] Matthew Green, Mathias Hall-Andersen, Eric Hennenfent, Gabriel Kaptchuk, Benjamin Perez, and Gijs Van Laer. Efficient proofs of software exploitability for real-world processors. *Proceedings on Privacy Enhancing Technologies*, 2023.

[34] Angèle M. Hamel, Jean-Charles Grégoire, and Ian Goldberg. The mis-entropists: New approaches to measures in Tor. CACR Technical Report 2011-18, 2011.

[35] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[36] Martin Hirt, Christoph Lucas, Ueli Maurer, and Dominik Raub. Graceful degradation in multi-party computation. In *ICITS*, pages 163–180, 2011.

[37] Matthew Humphries. AnyDVD now rips UHD Blu-rays with ‘unbreakable’ AACS 2.0. *PC Magazine*, 22 Dec. 2017.

[38] Rob Jansen, Florian Tschorsch, Aaron Johnson, and Björn Scheuermann. The sniper attack: Anonymously deanonymizing and disabling the Tor network. In *NDSS*, 2014.

[39] Nerla Jean-Louis, Yunqi Li, Yan Ji, Harjasleen Malvai, Thomas Yurek, Sylvain Bellemare, and Andrew Miller. Sgxonerate: Finding (and partially fixing) privacy flaws in TEE-based smart contract platforms without breaking the TEE. *Proceedings on Privacy Enhancing Technologies*, 2024.

[40] Aaron Johnson, Chris Wacek, Rob Jansen, Micah Sherr, and Paul Syverson. Users get routed: Traffic correlation on Tor by realistic adversaries. In *CCS*, pages 337–348, 2013.

[41] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: Epid provisioning and attestation services. *White Paper*, 1(1-10):119, 2016.

[42] Ari Juels and Ronald L Rivest. Honeywords: Making password-cracking detectable. In *CCS*, pages 145–160, 2013.

[43] Sanket Kanjalkar, Joseph Kuo, Yunqi Li, and Andrew Miller. Short paper: I can’t believe it’s not stake! resource exhaustion attacks on PoS. In *FC*, pages 62–69, 2019.

[44] Mahimna Kelkar, Soubhik Deb, Sishan Long, Ari Juels, and Sreeram Kannan. Themis: Fast, strong order-fairness in byzantine consensus. In *CCS*, pages 475–489, 2023.

[45] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for Byzantine consensus. In *CRYPTO*, pages 451–480, 2020.

[46] Aggelos Kiayias and Qiang Tang. How to keep a secret: Leakage deterring public-key cryptosystems. In *CCS*, pages 943–954, 2013.

[47] Oliver Knight. Ethereum bot gets attacked for $20m as validator strikes back. *Coindesk*.

https://www.coindesk.com/business/2023/04/03/ethereum-mev-bot-gets-attacked-for-20m-as-validator-strikes-back/.

[48] Joshua A Kroll, Joe Zimmerman, David J Wu, Valeria Nikolaenko, Edward W Felten, and Dan Boneh. Accountable cryptographic access control, 2018. https://www.cs.yale.edu/homes/jf/kroll-paper.pdf.

[49] Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Accountability: definition and relationship to verifiability. In *CCS*, pages 526–535, 2010.

[50] Albert Kwon, Mashael AlSabah, David Lazar, Marc Dacier, and Srinivas Devadas. Circuit fingerprinting attacks: Passive deanonymization of tor hidden services. In *USENIX Security*, pages 287–302, 2015.

[51] Jure Leskovec and Eric Horvitz. Planetary-scale views on a large instant-messaging network. In *WWW*, pages 915–924, 2008.

[52] Zihao Li, Jianfeng Li, Zheyuan He, Xiapu Luo, Ting Wang, Xiaoze Ni, Wenwu Yang, Xi Chen, and Ting Chen. Demystifying DeFi MEV activities in Flashbots bundle. In *CCS*, page 165–179, 2023.

[53] Akaki Mamageishvili, Mahimna Kelkar, Jan Christoph Schlegel, and Edward W. Felten. Buying time: Latency racing vs. bidding for transaction ordering. In *AFT*, pages 23:1–23:22, 2023.

[54] Akshaya Mani, T. Wilson-Brown, Rob Jansen, Aaron Johnson, and Micah Sherr. Understanding tor usage with privacy-preserving measurement. In *IMC*, page 175–187, 2018.

[55] Moxie Marlinspike. Technology preview: Private contact discovery for signal, Sep 2017.

[56] Joachim Neu, Ertem Nusret Tas, and David Tse. The availability-accountability dilemma and its resolution via accountability gadgets. In *FC*, pages 541–559, 2022.

[57] Joachim Neu, Ertem Nusret Tas, and David Tse. Short paper: Accountable safety implies finality. Cryptology ePrint Archive, Paper 2023/1301, 2023.

[58] Office of Public Affairs. Justice department sues google for monopolizing digital advertising technologies. https://www.justice.gov/opa/pr/justice-department-sues-google-monopolizing-digital-advertising-technologies, 2023.

[59] Jim O'Leary. Improving first impressions on signal, Nov 2021.

[60] Rafael Pass, Elaine Shi, and Florian Tramer. Formal abstractions for attested execution secure processors. In *EUROCRYPT*, pages 260–289, 2017.

[61] Tor Project. https://git.torproject.org/tor.git.

[62] Kaihua Qin, Liyi Zhou, and Arthur Gervais. Quantifying blockchain extractable value: How dark is the forest? In *IEEE S&P*, pages 198–214, 2022.

[63] Florentin Rochet and Olivier Pereira. Waterfilling: Balancing the Tor network with maximum diversity. *Proc. Priv. Enhancing Technol.*, 2017(2):4–22, 2017.

[64] Maayan Roth, Assaf Ben-David, David Deutscher, Guy Flysher, Ilan Horn, Ari Leichtberg, Naty Leiser, Yossi Matias, and Ron Merom. Suggesting friends using the implicit social graph. In *KDD*, page 233–242, 2010.

[65] Bruce Schneier. Attacking Tor: how the NSA targets users' online anonymity. *The Guardian*, 4:7, 2013.

[66] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[67] Peiyao Sheng, Gerui Wang, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath. Bft protocol forensics. In *CCS*, pages 1722–1743, 2021.

[68] Lance Spitzner. *Honeypots: tracking hackers*, volume 1. Addison-Wesley Reading, 2003.

[69] Signal Support. Backup and restore messages, [n. d.]. https://support.signal.org/hc/en-us/articles/360007059752-Backup-and-Restore-Messages.

[70] Signal Support. Linked devices, [n. d.].

[71] Signal Support. Re-registering using your signal pin, [n. d.].

[72] Signal Support. Signal pin, [n. d.].

[73] Signal Support. Troubleshooting multiple devices, [n. d.].

[74] Signal Support. What do i do if my phone is lost or stolen?, [n. d.].

[75] Florian Tramer, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, and Elaine Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *EuroS&P*, pages 19–34, 2017.

[76] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *USENIX ATC*, pages 645–658, 2017.

[77] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

[78] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue inflight data load. In *IEEE S&P*, pages 88–105, 2019.

[79] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. *CoRR*, abs/2006.13353, 2020.

[80] Stephan van Schaik, Alex Seto, Thomas Yurek, Adam Batori, Bader AlBassam, Christina Garman, Daniel Genkin, Andrew Miller, Eyal Ronen, and Yuval Yarom. SoK: SGX.Fail: How stuff get eXposed. 2022.

[81] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. In *CCS*, page 2421–2434, 2017.

[82] Ben Weintraub, Christof Ferreira Torres, Cristina Nita-Rotaru, and Radu State. A flash(bot) in the pan: measuring maximal extractable value in private pools. In *IMC*, page 458–471, 2022.

[83] Liam 'Akiba' Wright. Silent security scandal or dying profession? DeFi bug bounty wall of shame has millions in unpaid bounties. *CryptoSlate*, 17 Aug. 2023.

[84] Sen Yang, Kartik Nayak, and Fan Zhang. Decentralization of ethereum's builder market, 2024.

A Deferred Formalism Details

Challenge with generic soundness amplification. To see multiple rounds do not generically amplify soundness, consider the following standard application technique as a strawman: Judge asks the prover $\mathcal{P}$ to run the sting protocol ℓ times and submit its outputs. Taking the majority outcome as the output of Judge allows amplification of a sting protocol with ε-soundness to one with ε^*-soundness where $\varepsilon^* = \Pr[X > \ell/2]$ for $X \sim \mathsf{Binomial}(\ell, \varepsilon)$. This would give negligible soundness error with κ rounds.

However, a significant obstacle is that a malicious *prover $\mathcal{P}$ might run more than ℓ rounds and only submit the attempts where it succeeds*. If Judge cannot ensure that $\mathcal{P}$ submits all rounds, a malicious $\mathcal{P}$ can fool Judge.

In fact, any non-negligible soundness error implies a soundness error as high as ≈ 1. This is because the soundness definition considers all possible $\mathcal{P}^*$, including one that runs a given $\mathcal{P}'$ which fools Judge with $\varepsilon_s(\kappa)$ probability $\mathsf{poly}(\kappa)$ times (since the Judge algorithm is public and can be run internally by $\mathcal{P}^*$ to pre-decide whether the attempt will succeed). This will result in a soundness error of $\varepsilon_s(\kappa)$ immediately implying a soundness error of $1-(1-\varepsilon_s(\kappa))^{\mathsf{poly}(\kappa)}$ for any polynomial $\mathsf{poly}(\kappa)$. As a consequence, valid sting protocols will need to have at most negligible soundness error.

Fortunately, when designing our sting protocols in particular, we can amplify soundness by enforcing sequentiality in a non-black-box way. Recall that our protocols will typically work by first requesting a *stinger* input from a trusted "Sting Enclave". The final output is provided to the enclave again for a validity check on the proof. Here, we can enforce sequentiality in an ℓ-round proof by allowing the enclave only after the i-th round is completed to generate a stinger for the $i+1$-st round of the same proof instance.

B Deferred Protocol Details

Proof of Lemma 1. Our proof uses game-based formulation. Let $\mathbf{G}_0$ be the stinger soundness game for χ^*, F (see Definition 3). Let $\mathbf{G}_1$ be the same game except that the secret s is sampled from the uniform distribution $U_{|\chi^*|}$. Consider any $\mathcal{B}$ playing these game. We define $\mathsf{Adv}^{\chi^*,F}_{\mathsf{snd}}(\mathcal{B}) = \Pr[\mathcal{B} \text{ wins in } \mathbf{G}_0]$ as and $\mathsf{Adv}^{\chi^*}_{\mathsf{inv}}(\mathcal{B}) = \Pr[\mathcal{B} \text{ wins in } \mathbf{G}_1]$.

Now, consider a game pair $(\mathbf{G}^*_0, \mathbf{G}^*_1)$ for distinguishing between the distributions $D_0 = \chi^*$ and $D_1 = U_{|\chi^*|}$. Define an adversary $\mathcal{C}$ for this game pair as follows: $\mathcal{C}$ runs $\mathcal{B}$ internally and simulates $\mathbf{G}_0/\mathbf{G}_1$ for it (this depends on what $\mathcal{C}$ receives from its own game). Observe that $\mathcal{C}$ outputs 1 in $\mathbf{G}^*_b$ iff. $\mathcal{B}$ wins in $\mathbf{G}_b$ for $b \in \{0,1\}$. Define d_{χ^*} to be this distinguishing advantage $\left|\Pr[\mathcal{C} \text{ outputs 1 in } \mathbf{G}^*_0] - \Pr[\mathcal{C} \text{ outputs 1 in } \mathbf{G}^*_1]\right|$. Consequently,

$$\mathsf{Adv}^{\chi^*,F}_{\mathsf{snd}}(\mathcal{B}) \le d_{\chi^*} + \mathsf{Adv}^{\chi^*}_{\mathsf{inv}}(\mathcal{B})$$

Now, the maximum advantage for distinguishing between χ^* and $U_{|\chi^*|}$ (with 1 sample), even for unbounded adversaries is the total variation distance (tvd) between the distributions—this corresponds to guessing based on the maximum likelihood estimate. In other words, the above d_{χ^*} is bounded by $tvd(\chi^*, U_{|\chi^*|})$, which completes the proof.

Proof of Theorem 2. Recall that the completeness and soundness errors for 1 round are bounded by $p = tvd(\chi, \chi^*)$ and $r = tvd(\chi^*, U_{|\chi^*|})$ respectively. In other words, in one round, an honest prover succeeds with probability $1-p$, and an adversarial prover succeeds with probability r.

Now, intuitively, when $1-p > r$ (i.e., $1-p-r > 0$), by running more rounds, we can make the honest prover succeed with probability close to 1, and the adversarial prover succeed with probability close to 0.

Specifically, consider running n rounds such that the prover is required to succeed in more than $k = n(1-p+r)/2$ rounds

for the proof to be valid. We can calculate the success probability for both the honest and adversarial provers, using a binomial Chernoff bound which we recall below.

Fact 1 (Chernoff Bound)**.** Consider $Y \sim \mathsf{Binomial}(n, \rho)$ and let $\mathsf{W}(n, \rho, y) = \Pr[Y \leq y]$. If $y < np$, then $\mathsf{W}(n, \rho, y) \leq exp(-\frac{(n\rho - y)^2}{3n\rho})$.

Observe now that

$$
\begin{aligned}
&\Pr[\text{Honest prover fails—that is, it succeeds in at most } k \text{ rounds}] \\
&\leq exp\left(\frac{-(n(1-p) - n(1-p+r)/2)^2}{3n(1-p)}\right) \\
&\leq exp\left(\frac{-(n/2)^2(1-p-r)^2}{3n(1-p)}\right) \leq exp\left(\frac{-n(1-p-r)^2}{12(1-p)}\right)
\end{aligned}
$$

If we want to bound this error by ε_c, we get

$$n > \frac{-12\ln(\varepsilon_c)(1-p)}{(1-p-r)^2}$$

A similar calculation can be done with the probability that an adversarial prover fails in at most $n-k$ rounds, which is $\mathsf{W}(n, 1-r, n-k)$—this is the soundness error, which if we want to bound by ε_s results in

$$n > \frac{-12\ln(\varepsilon_s)(1-r)}{(1-p-r)^2}.$$

This completes the proof.

B.1 Remarks

The double-TEE setting. Over the years, a number of vulnerabilities [10, 12, 18, 78–81] have been identified within TEEs such as Intel SGX and although corrective patches have been applied to known issues, the emergence and exploitation of new vulnerabilities remain a persistent concern. This fact may cause concern over whether it is reasonable to use a TEE for our sting protocols.

In the specific case where both the Sting Enclave and the target application use the same type of TEE (e.g., Intel SGX)—what we call the *double-TEE* setting—sting proofs can detect an application vulnerability *even in the face of a TEE break*.

Observe that in the double-TEE setting, a sting proof shows one of two things: (1) Secrets are leaked by some subversion functionality O that may or may not involve a broken (application) TEE or (2) The TEE for the Sting Enclave has been broken. Either case would likely result in the target application's owner to revisit its TEE usage. Specifically, in case (2), if the TEE for the Sting Enclave has been broken, then because the same type of TEE is used for the target application, it could also be vulnerable to subversion. In essence, such a sting proof will rightly deter the use of the TEE within the target application as well.

Narrowing down the cause of failure. While sting protocols can capture a wide range of failures (e.g., implementation bugs to collusion to side-channel attacks), a basic limitation of our approach is that it can be difficult to pinpoint the exact security failure that gives rise to a subversion oracle. Still, in practice, external evidence can help. For example, if the target service has been formally verified and unlikely to fail, it might be possible to ascribe the problem to corrupted players.

C Deferred Details on Applications

C.1 Block Building

Protocol subtleties. An efficiency hurdle is how to enable the prover to identify s among the transactions leaked by O. Of course, the prover could attempt its sting proof with all transactions but this could be practically infeasible. One simple strategy to avoid this is to have the prover commit to the entire leakage (through e.g., a hash of it) when constructing t. Later, when opening the commitment, the prover can additionally prove to the Sting Enclave that it knows the preimage (i.e., all the leaked transactions) of this hash.

A better approach is actually for the prover to, before constructing t, input all leaked transactions to the Sting Enclave which identifies s if it exists within the list. Observe that this still does not prove to the Sting Enclave that the prover obtained s through O and not because of the fact that all transactions are eventually publicly revealed—this will be done by our technique of constructing a new transaction t by embedding information from s into it.

An additional question is on the types of subversion oracles we can handle—particularly relevant due to the time-based nature of our setting. For instance, if O only leaks the bundle transactions very close to public release of the block, then our sting protocol would not be able to be completed in time.

Still, we note that such leakage may not have significant practical relevance. For example, O could be because of a dark marketplace where the leaked bundle information is sold to the highest bidder. Here, if the leakage is very late, his would also mean that the e.g., "buyer" of this information would also have minimal time in order to exploit the leakage.

Analysis. Assume that our use of the default searcher code to construct stinger bundles makes the sting protocol undetectable to the SGX-builder. Then, observe that the sting protocol will succeed whenever places a competitive bid in the searcher-builder auction to get the newly crafted transaction t included.

On the soundness side, if SGX is secure, then a malicious prover will be able to fool the Sting Enclave only when it correctly guesses s—even if the transactions are sampled have low entropy, the transaction signature will provide enough entropy to prevent guessing s.

Recall that even when SGX is broken, since we are in the *double TEE* setting (where the target application is also in a

TEE; see Section 3), the sting proof should provide sufficient evidence to forgo the use of SGX in the builder application.

C.2 Signal Contact Discovery

Sting protocol details. In more detail, the full sting protocol is now as follows: The sting prover $\mathcal{P}$ supplies a phone number N (that it owns), and a contact list L to the Sting Enclave, which runs a Signal client within it. The enclave now randomly selects a portion (half, for simplicity) of the contacts in L to create a new contact list L' and registers a new Signal account $\mathcal{U}$ using N and L'—looking ahead, this corresponds to the secret s whose leakage will be considered.

Now, $\mathcal{P}$ can use its access to a subversion oracle O which leaks the secret L' (again, for simplicity, we consider O that leaks the entire contact list; partial leakage can be handled as before). $\mathcal{P}$ can now show this to the Sting Enclave, which simply checks whether it is the same as the original list L'. As standard, upon verification, the Sting Enclave provides an attestation, which can be checked publicly by a deterministic Judge contract.

Analysis and discussion. Unlike our previous examples, the actions of other users do not affect the creating of a sting proof. We can therefore focus on a single user $\mathcal{P}$, and a subversion oracle O that leaks its contact list.

To analyze soundness, we note that the probability of $\mathcal{P}$ perfectly guessing L' on its own is $1/\binom{n}{n'}$ where n, n' are the lengths of the lists L, L' respectively. Asymptotically, even choosing $n' = \log n$ results in a soundness error of $O(n^{\log n})$ which is negligible in n. In practice however, it might be best to, given n, choose an explicit n' such that $(1/\binom{n}{n'})$ is less than the desired soundness error ε; a typical choice could be $n' = n/2$ (which minimizes the explicit soundness error). As one concrete example, starting with $n = 90$ contacts, and choosing half of them to form L' already gives a soundness error smaller than 2^{-80}.

To completely capture soundness, we also need to ensure that a malicious prover cannot obtain L' through a different route—e.g., by attempting to re-register the account, or by linking additional devices. For this, we observe that Signal has a feature that enables requiring a PIN for these actions. This means that by having the Sting Enclave set the PIN for the account, a malicious prover cannot obtain L' in this fashion. More details on the soundness are provided in the subsequent paragraph. Finally, as alluded to in Section 3.1, although the above analysis assumes that SGX is secure, even if the prover can break privacy of the SGX Sting Enclave to retrieve L', in practice, this can serve as sufficient evidence for Signal to not use SGX for its application.

For completeness, we note that if the sting protocol is undetectable (i.e., it is indistinguishable from a "real" user's registration), the protocol will always be successful in showing the existence of an O that leaks the contact list.

For boosting soundness or completeness, unlike in our previous applications, it may be challenging to repeat the protocol multiple times. For instance, repeating registration with the same phone number many times can affect undetectability by e.g., triggering anti-spam policies of the platform [59]. Moreover, using different phone numbers or contact lists for each attempt could be prohibitively expensive since phone-numbers are a limited, hard-to-acquire resource. Still, we observe that if boosting is required, it can also be achieved by multiple provers independently carrying out the sting protocol. Furthermore, it might also be possible to partition the original contact list L into separate chunks for use in each iteration.

Soundness details. If SGX is not broken, a malicious U cannot use the control over N in any way to break soundness by learning L', without alerting the Sting Enclave. To show this, we describe the Signal account management. Suppose that π creates the Signal account $\mathcal{U}_{Signal}$ with the phone number provided by the U. After this, a malicious U could access $\mathcal{U}_{Signal}$'s information using a backup, re-registering the number N, transferring the account, or adding a new linked device [69–71, 73, 74].

The backups are encrypted, stored locally, and disabled by default [69]. Hence, π would not to enable backups and U would be unable to acquire a backup. To re-register a number in a new device and recover the previous contacts and settings, U would need a PIN created in π [71, 72]. Such a PIN can be a long alphanumeric string, so the SGX would construct a PIN that is hard to guess. Even if U guessed the PIN, and successfully re-registered N into their own device, this action would be visible to the SGX, since its client would now be unregistered [71]. If that was the case, it would refuse to create a signature σ. To transfer an account, it is necessary to have access to either the main device (inside the SGX), or to a backup [69]. Therefore, U would not be able to obtain information with this method. Finally, to link a new device, there needs to be consent from the main device, which is controlled by the Sting Enclave, so this is not an option either [70].

In summary, if SGX is not compromised, U cannot obtain L' without it being noticeable to the Sting Enclave, which would cause the protocol to fail in producing a proof. Hence, if U acts maliciously, it can break soundness by learning L' only with negligible probability, from guessing L' at random.

C.3 Transaction Ordering

Extension to a distributed setting. We first observe that our approach directly works even in a distributed setting for leader-based protocols where the current leader controls the transaction ordering in the current block. The extension for distribution ordering protocols (such as causal and fair ordering) is also straightforward. The sting prover will now submit transaction A to all servers, and commit to receipts from a quorum of nodes within its second transaction B. Now, once

the two transactions are ordered, the prover can reveal its commitment, to show whether or not according the ordering was consistent to the claimed policy.

Sting infrastructure as part of the server design. As a concrete example, consider a single centralized server which sequences (and executes) transactions from users. Consider the following design as providing protection against censorship and manipulation of the ordering by the server: The server houses an SGX *Sting Enclave* which will submit "stinger" transactions to the server itself. The enclave observe the final output sequence to ensure that the transactions that it submitted are present in the correct ordering. In essence, if the server cannot distinguish between real user transactions, and stinger transactions, then with high probability, if it attempts to censor or reorder transactions, this will also impact the stinger transactions and will be detected by the Sting Enclave. This will provide convincing proof that the server has manipulated the transaction ordering.

C.4 Auction Setting

Consider the setting of a second-price auction—after bidders submit bids to the auctioneer, the highest bid wins the item but only pays the second-highest bid. Second-price auctions are truthful—that is, the dominant strategy for a bidder is simply to bid its value regardless of the other's strategy. One concern is that in settings with many or unknown bidders, the auctioneer can itself pose itself as a bidder, and only slightly undercut the highest bidder to maximize its revenue. This poses a problem if the auctioneer is centralized even if bids are sealed, and not disclosed to all bidders. In fact, there is evidence of similar behavior in the real world as brought to light by the Justice department's lawsuit against Google for unfairly manipulating its ad auctions [58].

A simple sting protocol is possible for detecting malicious auctioneer behavior. Suppose that the prover knows that the auctioneer strategy is to add an additional bid $F(b)$, where b is the highest bid. Our techniques from Section 3 already construct a sting proof when there is a subversion oracle that leaks b, and thus apply to this malicious-auctioneer scenario.

Sting proof without leakage. Remarkably, a sting proof is also possible *without any leakage*—we provide a sketch below. The intuitive idea is to use the Sting Enclave to generate a bid, and win the auction in a way that the bidder's payment could only have been influenced by fake bid $F(b)$.

First, the prover uses the Sting SGX Enclave to generate a bid b' for an auction instance so that b' is unknown to even the prover. With some probability, b' will win the auction, and $F(b')$ will be maliciously added by the auctioneer to make the prover pay $F(b')$ instead of the true second-highest bid in the auction. Since only the auctioneer knows the final bids, if $F(b)$ is the payment, then it is very likely that it was maliciously added by the auctioneer. This can be shown to the Sting Enclave to create a sting proof. Repeating this experiment many times will reduce the proof soundness error.

Bidder collusion. Recall that we consider a second-price auction setting—after bidders submit sealed bids to the auctioneer, the highest bidder wins the item but only pays the second-highest bid. For simplicity, suppose that we have only two bidders. Recall that second-price auctions are truthful—that is, the dominant strategy for a bidder is simply to bid its value regardless of the other's strategy. However, it is well-known that bidder collusion can make the auctioneer revenue zero; this is done by having one party bid zero. Sting protocols provide one way to fix this: Communication between parties can serve as evidence of collusion and the auctioneer (or mechanism) can now accept such a proof to penalize participants. For instance, if Alice communicates with Bob to initiate the collusion, Bob can prove this to the auctioneer so that he collects a reward and Alice is penalized. We leave a full game-theoretic exploration of this idea for different auction settings to future work.

D Deferred Details on Implementation and Evaluation

In this section, we provide the deferred sting protocol code for our Tor and block-building applications.

Algorithm 2 Sting Protocol for Tor

```
1: ProvisionEnclave(C = {DAs, rep, destSet}):                ▷ Sting Enclave
2:     ...
3:     relaySet = HttpsRequest(DAs)
4:     Seal(sk_Sting, rep, relaySet, destSet)
5:     ...
6: GenStinger(C) → β:                                          ▷ Sting Enclave
7:     rep, relaySet, destSet = Unseal()
8:     S = ∅
9:     for i = 1 to rep do
10:        Sample stinger:
11:            circuit = (relay1, relay2, exit) ←$ relaySet
12:            dest ←$ destSet
13:        Access dest using circuit via target application (Tor)
14:        Leak sid (circuit's unique id in Tor) to prover
15:        S = S ∪ (sid, dest)
16:    Seal(rep, S)
17: Leak data {(sid′, dest′)} from target application        ▷ Subversion Oracle
18: VerifyEvidence(E={(sid′,dest′)})→(prf,sig_Sting):          ▷ Sting Enclave
19:    ...
20:    rep, S = Unseal()
21:    cnt = 0
22:    Verify(S, E):
23:        for (sid′, dest′) in E do
24:            if (sid′, dest′) ∈ S then cnt = cnt + 1
25:        return Satisfied(cnt, rep)
26:    GenProof(): prf = "OK"
27:    ...
28: CollectBounty(sig_Sting, prf, pk_Sting):                   ▷ prover
29:    ...
30:    VerifySig(sig_Sting, prf, pk_Sting): prf =? "OK"
31:    ...
```

Algorithm 3 Sting Protocol for Block-Building

```
1: GenStinger(C=(U({0,1}^256),S_unsigned,bundle)) → β:        ▷ Sting Enclave
2:     Sample ECDSA signature nonce: k ←$ U({0,1}^256)
3:     pk_stinger, sk_stinger = EthKeyGen()
4:     S_signed = SignTx_{sk_stinger}(S_unsigned, nonce_ECDSA=k)
   *** CRITICAL SEQUENCE STARTS ***
5:     Send bundle||S_signed to target application
6:     Seal(k)
7:     Leak backdoor β=S.hash to prover
8: Leak data L (all bundles received) from builder            ▷ Subversion Oracle
9: MakeEvidence(L, β, C=T_unsigned) → E:                      ▷ prover
10:    Use β=S.hash to locate S in L
11:    Randomly sample commitment blinding factor r
12:    Make commitment com = Com(S.sig, r)
13:    pk_evidence, sk_evidence = EthKeyGen()
14:    T_signed = SignTx_{sk_evidence}(T_unsigned, nonce_ECDSA=com)
15:    Send T_signed to target application
   *** CRITICAL SEQUENCE ENDS ***
16:    Make evidence E = (r,block,S,T,sk_evidence)
17: VerifyEvidence(E=(r,block,S,T,sk_evidence)) → (prf, sig_Sting):   ▷ Sting Enclave
18:    ...
19:    Verify(S, E):
20:        Check both S and T are included in block
21:        T.sig =? SignTx_{sk_evidence}(StripSig(T),nonce_ECDSA=Com(S.sig,r))
22:    GenProof(): prf = block.num||block.hash
23:    ...
24: CollectBounty(sig_Sting, prf, pk_Sting):                   ▷ prover
25:    ...
26:    VerifySig(sig_Sting, prf, pk_Sting)
27:        blockNum, blockHash = parse(prf)
28:        blockhash(blockNum) =? blockHash
29:    ...
```