Sealed-Glass Proofs: Using Transparent Enclaves to Prove and Sell Knowledge

Florian Tramèr[1], Fan Zhang[2,4], Huang Lin[1], Jean-Pierre Hubaux[1], Ari Juels[3,4], Elaine Shi[2,4]
[1] EPFL, [2] Cornell University, [3] Cornell Tech, Jacobs Institute,
[4] Initiative for CryptoCurrencies and Contracts (IC3)

ABSTRACT

Trusted hardware systems, such as Intel's new SGX instruction set architecture extension, aim to provide strong confidentiality and integrity assurances for applications. Recent work, however, raises serious concerns about the vulnerability of such systems to side-channel attacks.

We propose, formalize, and explore a cryptographic primitive called a *Sealed-Glass Proof (SGP)* that captures computation possible in an isolated execution environment with *unbounded leakage*, and thus in the face of arbitrarily powerful side-channel attacks. A SGP specifically models the capabilities of trusted hardware that can attest to *correct execution* of a piece of code, but whose execution is *transparent*, meaning that an application's secrets and state are visible to other processes on the same host.

Despite this strong threat model, we show that a SGP can support a range of practical applications. Our key observation is that a SGP permits safe verifiable computing in zero-knowledge, as information leakage results only in the prover learning her own secrets. Among other applications, we describe the implementation of an end-to-end bug bounty (or zero-day solicitation) platform that couples a SGX-based SGP with a smart contract. This platform enables a marketplace that achieves fair exchange, protects against unfair bounty withdrawals, and resists denial-of-service attacks by dishonest sellers. We also consider a slight relaxation of the SGP model that permits black-box modules instantiating minimal, side-channel resistant primitives, yielding a still broader range of applications. Our work shows how trusted hardware systems such as SGX can support trustworthy applications even in the presence of side channels.

1. INTRODUCTION

Trusted hardware platforms aim at creating secure and isolated software execution environments that could lead to many practical applications of secure multiparty computation. For instance, Intel's newly released Software Guard Extensions [25, 26] (SGX) allows programmers to create secure *enclaves* that execute in isolation from the rest of a host's software, including its OS.

While these trusted platforms aim to protect the integrity, authenticity and confidentiality of enclaved programs against a variety of software or physical attacks [25, 26], the confidentiality goal appears elusive. Recent work has shown that enclaves may leak large amounts of sensitive information to a malicious host through their memory access patterns [49]. Additional software or physical side-channel attacks can further compromise an enclave's secrets [17]. Such attacks are in no way unique to the SGX platform, and a variety of countermeasures have been developed over the years [18, 47, 51], usually to protect highly sensitive cryptographic code. It is reasonable to assume that trusted hardware platforms can successfully apply these techniques to achieve strong protection for a limited set of cryptographic operations using long term secrets (e.g., cryptographic keys used for attesting to an enclave's contents or for protecting enclave memory). Such protection is essential: A successful extraction of a platform's private key would essentially translate into a total security break. A much more challenging goal is extending side-channel protections to arbitrary computations [18, 22, 32, 38, 39] as the resulting performance degradation is non-negligible [17, 38, 41] and it remains unclear whether data leaks can truly be fully eliminated.

We thus initiate an investigation of a model of secure computation that is intentionally weaker than that of traditional trusted hardware. Our goal is to predicate such computation on security assumptions that are more easily achievable in practice. We define a formal model of *transparent enclave execution* in which enclaves guarantee execution integrity and authenticity, but not confidentiality. That is, we assume that the trusted hardware's attestation and integrity protection mechanisms are secure, but that a program running in an enclave *leaks unbounded information* to the host.

The question immediately arises: *Is this weaker model of trusted hardware useful?* Our key insight is that transparent enclaves provide sufficient guarantees to securely realize a number of functionalities characterized by a "unidirectional" *resource asymmetry* between untrusted parties, for example when one player has knowledge of a secret, but not the other. Indeed, by having the party holding the secret host the enclave, we forego the need for execution confidentiality and can rely solely on execution integrity and authenticity. To demonstrate the usability of our model, we show how to efficiently realize a number of interesting primitives in the UC framework [12], such as *Verifiable Computing*, *Commitment Schemes*, and *Zero-Knowledge Proofs*.

We go further to show that transparent enclaves can securely realize a powerful, general functionality that encompasses the above three primitives. We call this functionality a *sealed-glass proof* (SGP). SGPs are similar in spirit to the "commit-and-prove" functionality of [15]: A prover first *seals* (i.e., commits to) a program and input in an enclave, and later runs the program over her own input in addition to a "challenge" input provided by a verifier. We further show that in cases where the verifier's challenge is simply a random string, the enclave environment can be leveraged to make such "seal-and-randomize" schemes non-interactive.

When built on top of SGX, SGPs enable provers to convince verifiers of the execution of standalone programs written in standard C or C++ [26]. Prior work further shows that SGX-based SGPs can be applied to *unmodified legacy*

applications [5] as well as Map-Reduce computations [19].

A compelling application of SGPs is *fair exchange* of a secret for a monetary reward. In their full generality, such transactions have a number of applications, such as the sale of 0-day exploits (as in a bug bounty system) and payments contingent on a proof of identity. A recently proposed scheme by Maxwell [33] known as Zero-Knowledge Contingent Payments (ZKCP) enables Bitcoin transactions for the sale of information between mutually untrusted parties. As we show, however, this scheme and an extension in [3] have serious limitations in terms of efficiency, requirements for interaction among parties, and vulnerability to buyers retracting their offers and denying payment to honest sellers.

We show that SGPs combined with *smart contracts* [42] can overcome these limitations and thus vastly improve the safety and efficiency of such fair exchange protocols in *digital knowledge marketplaces* where buyers and sellers are able to launch respectively *bounties* or *auctions* for secret resources. For buyer-initiated transactions, we further describe slight relaxations to our transparent enclave execution model that protect against unfair withdrawal of a bounty, a property missing in the original ZKCP protocol.

To demonstrate the practicality of such knowledge marketplaces, we design and implement a secure and fair bug-bounty system on top of SGX and Ethereum [11, 48]. We describe use cases of our system involving the sale of 0-day exploits in several real-world applications: (1) A SQL injection vulnerability in a web login form; (2) Disparities in certificate validation logic in different TLS implementations; and (3) A generic framework for MITM attacks that encompasses a wide range of attacks on TLS handshakes.

Contributions. In summary, our work offers the following contributions:

(1) We introduce the novel notion of *transparent enclave execution*, a model of trusted hardware that captures unbounded leakage of application data and thus arbitrarily powerful side-channel attacks. We additionally formalize slight relaxations of this strong threat model that expand the range of applications for which secure realization of efficient protocols is possible.

(2) We formalize and show how to instantiate *Sealed-Glass Proofs* (SGPs), a primitive realizable with transparent enclaves that encompasses and generalizes verifiable computing, commitment schemes, and ZK proofs.

(3) We present protocols that use SGPs coupled with smart contracts to realize *knowledge marketplaces* with strong fairness guarantees. We describe a practical implementation of such a marketplace in the form of a secure and fair bug-bounty platform built on top of SGX and Ethereum, and report on experiments with sample proofs of 0-day exploits.

2. PRELIMINARIES

In this section, we formalize our novel model of transparent enclave execution and discuss its relevance in view of known side-channel attacks against current trusted hardware platforms. We further describe generic multiparty computation settings in which our model proves useful.

2.1 Transparent Enclave Execution

The model we describe here serves as a relaxation of the standard model of trusted hardware (e.g., as in SGX), in which a program's execution is assumed to be fully concealed from the host. In our model, only a minimal number of critical functionalities (e.g., sensitive crypto code such as random number generators or the platform's attestation mechanism) are assumed to be kept secret from the host, while software enclaves are only assumed to provide authenticity and integrity when executing arbitrary user programs.

In the same sense that enclaves serve as a separation between *trusted* code and *untrusted* code, we further distinguish between tasks that require *confidentiality* from tasks that require only *authenticity* and *integrity*. This lets us side-step many critical and practical concerns about side-channel attacks arising from the execution of arbitrary code on platforms such as SGX [49]. Our thesis is that in practice, *only a small number of privileged cryptographic functionalities will be successfully concealed from the host*, through the use of well-studied cryptographic side-channel defenses (e.g., blinding, code randomization, leakage resilience) [51].

We begin by introducing some notation used throughout this work, borrowed mainly from [38].

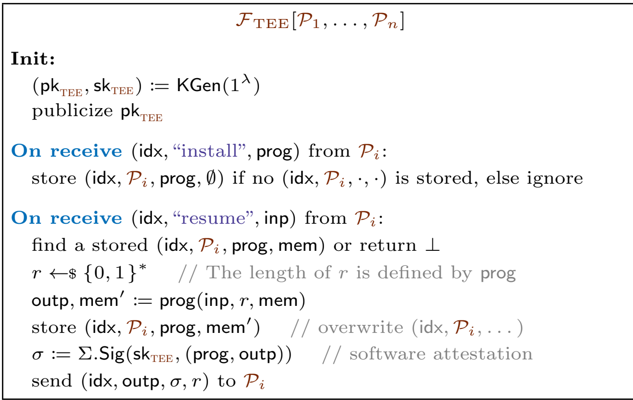
$\mathcal{P}$	The host/prover equipped with trusted hardware
$\mathcal{V}$	Identifier of the verifier
prog	A program to run in an enclave
$\mathsf{inp}, \mathsf{outp}$	The program's input and output
mem	The program's memory tape
$\mathsf{sid}, \mathsf{idx}$	Session identifier and Enclave identifier
λ	Security parameter
Σ, KGen	A Signature Scheme and its key generation algorithm
$\leftarrow\$$	Uniformly random sampling from the right-hand set

Formal Model.

For our purposes, enclaves are isolated software containers loaded with particular programs. We modify the idealized trusted hardware abstraction of Shi et al. [38, 50], to account for our relaxed enclave transparency assumptions. Our hope is that our new abstraction narrows the gap between the idealized functionality and real-world implementations of trusted hardware platforms. Following [38, 50], we define an ideal functionality $\mathcal{F}_{\text{TEE}}$ (Transparent Enclave Execution) for a trusted hardware platform used by any number of hosts $\mathcal{P}$ (Figure 1). Our functionality abstracts a software enclave as a third party trusted for execution integrity and authenticity, but not confidentiality, with respect to $\mathcal{P}$.

In our context, a host (or prover) $\mathcal{P}$ wants to prove statements about the execution of some arbitrary program prog to a remote client (the verifier) $\mathcal{V}$. The program gets access to a trusted source of randomness, but its execution is entirely transparent to the host (i.e., prog and all its inputs including the consumed randomness are revealed to $\mathcal{P}$). Only the platform's attestation mechanism (more specifically a secret signing key sk_{TEE}) is assumed to remain hidden from the host. This model captures a setting in which the execution of arbitrary code in an enclave leaks *complete information* about that program's memory to the host, e.g., through various side channels. This setting can be generalized by considering various constrained information leakage models.

As described in Figure 1, $\mathcal{P}$ can load a program prog into an enclave with identifier idx. Upon a "run" call with input inp, prog is executed on inp and fresh randomness r until it produces an output outp. All the program's inputs and outputs are revealed to $\mathcal{P}$. The "resume" call of $\mathcal{F}_{\text{TEE}}$ can capture a setting in which prog is some *stateful* program that relies on $\mathcal{P}$ to handle requests for addi-


$\mathcal{F}_{\text{TEE}}[\mathcal{P}_1, \ldots, \mathcal{P}_n]$

Init:
$(\mathsf{pk}_{\text{TEE}}, \mathsf{sk}_{\text{TEE}}) := \mathsf{KGen}(1^\lambda)$
publicize pk_{TEE}

On receive $(\mathsf{idx}, \text{“install”}, \mathsf{prog})$ from $\mathcal{P}_i$:
store $(\mathsf{idx}, \mathcal{P}_i, \mathsf{prog}, \emptyset)$ if no $(\mathsf{idx}, \mathcal{P}_i, \cdot, \cdot)$ is stored, else ignore

On receive $(\mathsf{idx}, \text{“resume”}, \mathsf{inp})$ from $\mathcal{P}_i$:
find a stored $(\mathsf{idx}, \mathcal{P}_i, \mathsf{prog}, \mathsf{mem})$ or return $\perp$
$r \leftarrow\$ \{0,1\}^*$ // The length of r is defined by prog
$\mathsf{outp}, \mathsf{mem}' := \mathsf{prog}(\mathsf{inp}, r, \mathsf{mem})$
store $(\mathsf{idx}, \mathcal{P}_i, \mathsf{prog}, \mathsf{mem}')$ // overwrite $(\mathsf{idx}, \mathcal{P}_i, \ldots)$
$\sigma := \Sigma.\mathsf{Sig}(\mathsf{sk}_{\text{TEE}}, (\mathsf{prog}, \mathsf{outp}))$ // software attestation
send $(\mathsf{idx}, \mathsf{outp}, \sigma, r)$ to $\mathcal{P}_i$

Figure 1: **Ideal transparent enclave execution.**

tional inputs (potentially from other parties). In most of this work, it will be sufficient to consider stateless programs that take in a single input inp and random string r and compute $\mathsf{outp} \leftarrow \mathsf{prog}(\mathsf{inp}, r)$. One situation in which we will make use of stateful transparent enclaves is in securely realizing UC commitments in Section 3. Note that the host can launch multiple enclaves with different idxs running the same program prog.

Enclave programs can generate attestations on each “resume” invocation (as is possible in SGX for instance [26]). An attestation is a digital signature, under a secret key sk_{TEE}, of the loaded program (i.e., a digest of the program binary) and the program's output outp.

Remarks about Universal Composability.

We prove the security of our protocols in the Universal Composability (UC) framework [12], to illustrate the strong security and modularity guarantees offered by our transparent enclave execution model. More specifically, $\mathcal{F}_{\text{TEE}}$ is the “multi-session extension” [16] of an ideal enclave functionality (i.e., a functionality that allows for a single enclave to be launched). If multiple protocols make use of trusted enclaves, these protocols implicitly share *joint state* in the form of the platform's attestation key sk_{TEE}. By the JUC Theorem [16], arbitrarily many *independent* protocols making use of trusted transparent enclaves can be securely composed with each other by making use of $\mathcal{F}_{\text{TEE}}$.

We will assume that each party $\mathcal{P}$ is identified by a unique id (simply denoted $\mathcal{P}$) and by a session id sid [12].

The proposed formalism of $\mathcal{F}_{\text{TEE}}$ currently fails to capture composition with *arbitrary* other protocols sharing a global trusted hardware functionality. Indeed, platforms such as SGX are becoming globally available, meaning that an arbitrary number of protocols may end up using a same shared state in the form of the platform's attestation key. Stronger composability guarantees could be obtained by proving security of our protocols in the GUC framework [13], by modeling $\mathcal{F}_{\text{TEE}}$ as a single global functionality. Extending the results in this paper to GUC introduces some technical artifacts, regarding the modeling of digital signatures in the GUC framework [**?**]. We defer a formal discussion of these matters, and of an appropriate formalism of $\mathcal{F}_{\text{TEE}}$ as a global functionality, to a forthcoming manuscript [**?**].

Relation to Intel's SGX.

SGX [26] allows a host to create multiple enclaves that run programs of the host's choosing. The “resume” call in the $\mathcal{F}_{\text{TEE}}$ functionality abstracts two mechanisms of SGX: (1) A host may call any trusted function exposed by the loaded program, or (2) A trusted program may explicitly call an untrusted function outside of the enclave. The latter can be seen as prog producing some intermediate output to be sent as a request to $\mathcal{P}$, and awaiting a “resume” callback with $\mathcal{P}$'s response. SGX enclaves further have access to a trusted hardware random number generator. The signature scheme in $\mathcal{F}_{\text{TEE}}$ models SGX's *quoting enclave* that authenticates and signs reports obtained from running enclaves. Attestations can be publicly verified by remote clients.

The transparent execution model of $\mathcal{F}_{\text{TEE}}$ highlights known vulnerabilities associated with the execution of arbitrary programs in SGX enclaves. Indeed, although SGX aims at protecting against known software and hardware attacks [26], it has been shown that enclaves may leak considerable information through side channels [17,49]. Our model thus pessimistically assumes that computations that have not been explicitly hardened against known side-channel attacks leak their complete execution trace to the host. We do rely on assumptions of execution integrity and secure attestation, however. As we shall see, these two properties actually suffice in a number of interesting applications.

2.2 Exploiting Resource Asymmetry

Given this model of transparent enclave execution, the question we seek to answer is: *“What kinds of protocols can we design that are secure in this model”*?

Our key observation is that transparent enclaves can be securely deployed in environments characterized by an inherent *resource asymmetry* between mutually untrusted parties, e.g., in terms of access to knowledge, computational power, or storage. We consider a generic setting in which a prover $\mathcal{P}$ wants to convince a verifier $\mathcal{V}$ that she ran a program prog on some input inp and obtained output outp. We describe three scenarios that correspond to different forms of resource asymmetry between provers and verifiers:

Verifiable Computing (VC): In a VC scheme, $\mathcal{P}$ convinces $\mathcal{V}$ that she correctly executed prog with input inp and output outp. The asymmetry lies in the resources committed by the two players: The benefit of VC is that $\mathcal{V}$ can obtain a correct outp while committing much less resource, e.g., computation, than $\mathcal{V}$. In our setting, a transparent enclave can be used to consume $\mathcal{P}$'s resources on behalf of $\mathcal{V}$.

Zero-Knowledge (ZK) Proofs: In a ZK proof, $\mathcal{P}$ wants to convince $\mathcal{V}$ that there is an input inp such that $\mathsf{prog}(\mathsf{inp}) = \mathsf{true}$, without leaking any information about inp. We are primarily interested here in zero-knowledge *proofs of knowledge*, in which $\mathcal{P}$ wants to prove that she knows such an inp. Here, there is a knowledge asymmetry, and an enclave can be used to check $\mathcal{P}$'s knowledge of inp on $\mathcal{V}$'s behalf.

Sealed-Glass Proofs (SGP): An SGP generalizes VC and ZK schemes. In many protocols for both schemes, $\mathcal{V}$ provides an *ephemeral secret* (i.e., a challenge) r to $\mathcal{P}$. As enclave execution is transparent to the prover in our model, enclaves cannot be trusted to safeguard r. We can use our model, however, to ensure that a prover *commits* a program and input of her choice before learning r. We describe applications of SGPs in the construction of a *digital knowledge marketplace*, and a fair bug-bounty program in particular.

$\mathcal{F}_{\text{SGP}}[\mathcal{P}, \mathcal{V}, \mathsf{prog}]$

On receive $(\mathsf{sid}, \text{"commit"}, \mathsf{inp}_{\mathcal{P}})$ from $\mathcal{P}$:

store $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}})$

send $(\mathsf{sid}, \text{"commit"})$ to $\mathcal{V}$

ignore subsequent $(\mathsf{sid}, \text{"commit"}, _)$ messages from $\mathcal{P}$

On receive $(\mathsf{sid}, \text{"proof"}, \mathsf{inp}_{\mathcal{V}})$ from $\mathcal{V}$:

find a stored $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}})$ or return $\perp$

let $\mathsf{outp} := \mathsf{prog}(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}})$

send $(\mathsf{sid}, \mathsf{inp}_{\mathcal{V}}, \mathsf{outp})$ to $\mathcal{P}$

On receive $(\mathsf{sid}, \text{"ok"})$ from $\mathcal{P}$:

send $(\mathsf{sid}, \text{"proof"}, \mathsf{outp})$ to $\mathcal{V}$

On receive $(\mathsf{sid}, \text{"open"})$ from $\mathcal{P}$:

find a stored $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}})$ or return $\perp$

send $(\text{"open"}, \mathsf{inp}_{\mathcal{P}})$ to $\mathcal{V}$

Figure 2: **Ideal Sealed-Glass Proof System.**

3. SEALED-GLASS PROOFS

Suppose you know an exploit against a system with a database (e.g., a SQL injection attack). You may want not to share this exploit explicitly, but instead just prove knowledge of it (e.g., to later sell it for a bug bounty). Directly demonstrating the exploit on the vulnerable system is not an option, of course, as it might then be learned by the system owner. Instead, if the system owner provided the full system environment, including the database, you could set up this environment in an enclave and prove the validity of the exploit by attesting to a valid program trace that violates the access-control policy (a NIZK proof, in essence). This approach is not an option either, though, as the database contents may be private and sensitive (and large).

In most cases, though, it would suffice to prove that your exploit works for *some* database. The system-owner could then instantiate an enclave you control with a "challenge" environment containing a database with randomly generated secrets. In a naïve approach to proving the validity of your exploit, you might input it into the enclave and show that it can output secrets from the "challenge." This would work if an enclave preserved confidentiality, but cannot work in a transparent execution environment: You could fake an exploit simply by extracting secrets from the enclave and embedding them in the exploit. To avoid such cheating by a prover, we can instead ask the prover to *commit* to her exploit before accepting a challenge from the system owner.

The $\mathcal{F}_{\text{SGP}}$ Functionality.

Sealed-Glass Proofs meet the requirements of the above scenario, permitting, e.g., a sound, zero-knowledge proof of knowledge of an exploit. Specified in Figure 2, an SGP may be viewed as combining commitment schemes with (zero-knowledge) verifiable computation. A prover commits to running a known program on her secret input, together with a challenge input provided by the verifier. The key idea is that the verifier's input is revealed only *after the prover commits to her own input.* The verifier obtains the output of the computation, and the prover may later share her secret input with the verifier by opening her commitment. As we show in Section 5, an SGP with an efficient decommitment mechanism can form the basis of a *digital knowledge marketplace* with strong guarantees for fair-exchange.

Prot$_{\text{SGP}}$

Prover $\mathcal{P}$: input $= (\mathsf{sid}, \mathsf{prog}, \mathsf{inp}_{\mathcal{P}})$

choose a fresh idx and send $(\mathsf{idx}, \text{"install"}, \widehat{\mathsf{prog}})$ to $\mathcal{F}_{\text{TEE}}$

send $(\mathsf{idx}, \text{"resume"}, (\mathsf{sid}, \text{"commit"}, \mathcal{P}, \mathsf{inp}_{\mathcal{P}}))$ to $\mathcal{F}_{\text{TEE}}$

On receive $(\mathsf{idx}, (\mathsf{sid}, \text{"commit"}, \mathcal{P}, N), \sigma_{\text{C}}, _)$ from $\mathcal{F}_{\text{TEE}}$:

send $(\mathsf{sid}, N, \sigma_{\text{C}})$ to $\mathcal{V}$

On receive $(\mathsf{sid}, \mathsf{inp}_{\mathcal{V}})$ from $\mathcal{V}$:

send $(\mathsf{idx}, \text{"resume"}, (\mathsf{sid}, \text{"proof"}, \mathsf{inp}_{\mathcal{V}}))$ to $\mathcal{F}_{\text{TEE}}$

On receive $(\mathsf{idx}, (\mathsf{sid}, \text{"proof"}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N), \sigma_{\text{P}}, _)$ from $\mathcal{F}_{\text{TEE}}$:

send $(\mathsf{sid}, \mathsf{outp}, \sigma_{\text{P}})$ to $\mathcal{V}$

// some time later...

send $(\mathsf{idx}, \text{"resume"}, (\mathsf{sid}, \text{"open"}))$ to $\mathcal{F}_{\text{TEE}}$

On receive $(\mathsf{idx}, (\mathsf{sid}, \text{"open"}, \mathsf{inp}_{\mathcal{P}}, N), \sigma_{\text{O}}, _)$ from $\mathcal{F}_{\text{TEE}}$:

send $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}}, \sigma_{\text{O}})$ to $\mathcal{V}$

Verifier $\mathcal{V}$: input $= (\mathsf{sid}, \mathsf{prog}, \mathsf{inp}_{\mathcal{V}})$

On receive $(\mathsf{sid}, N, \sigma_{\text{C}})$ from $\mathcal{P}$:

assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{C}}, (\widehat{\mathsf{prog}}, (\mathsf{sid}, \text{"commit"}, \mathcal{P}, N)))$

send $(\mathsf{sid}, \mathsf{inp}_{\mathcal{V}})$ to $\mathcal{P}$

On receive $(\mathsf{sid}, \mathsf{outp}, \sigma_{\text{P}})$ from $\mathcal{P}$:

assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{P}}, (\widehat{\mathsf{prog}}, (\mathsf{sid}, \text{"proof"}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N)))$

// some time later...

On receive $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}}, \sigma_{\text{O}})$ from $\mathcal{P}$:

assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{O}}, (\widehat{\mathsf{prog}}, (\mathsf{sid}, \text{"open"}, \mathsf{inp}_{\mathcal{P}}, N)))$

Figure 3: **Protocol realizing $\mathcal{F}_{\text{SGP}}$.**

Enclave program wrapper $\widehat{\mathsf{prog}}$

On input $((\mathsf{sid}, \text{"commit"}, \mathcal{P}, \mathsf{inp}_{\mathcal{P}}), r, \mathsf{mem})$:

parse r as a nonce N of length λ // $\mathsf{mem}' = \mathsf{inp}_{\mathcal{P}}||N$

return $(\mathsf{sid}, \text{"commit"}, \mathcal{P}, N)$ //will be included in σ_{C}

On input $((\mathsf{sid}, \text{"proof"}, \mathsf{inp}_{\mathcal{V}}), _, \mathsf{mem})$:

parse mem as $\mathsf{inp}_{\mathcal{P}}||N$

let $\mathsf{outp} := \mathsf{prog}(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}})$ // $\mathsf{mem}' = \mathsf{inp}_{\mathcal{P}}||N$

return $(\mathsf{sid}, \text{"proof"}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N)$ //will be included in σ_{P}

On input $((\mathsf{sid}, \text{"open"}), _, \mathsf{mem})$:

parse mem as $\mathsf{inp}_{\mathcal{P}}||N$

return $(\mathsf{sid}, \text{"open"}, \mathsf{inp}_{\mathcal{P}}, N)$ //will be included in σ_{O}

Figure 4: **The program wrapper used in Prot$_{\text{SGP}}$.**

Realizing $\mathcal{F}_{\text{SGP}}$ in the $\mathcal{F}_{\text{TEE}}$-Hybrid Model.

$\mathcal{F}_{\text{SGP}}$ is similar to the "commit-and-prove" functionality $\mathcal{F}_{\text{CP}}$ of Canetti et al. [15]. As they show, $\mathcal{F}_{\text{CP}}$ can be realized in the UC framework by combining a standard commitment scheme with an ideal ZK functionality.

The protocol we propose here makes use of $\mathcal{F}_{\text{TEE}}$ to directly realize both UC commitments and ZK proofs. This holds because $\mathcal{F}_{\text{TEE}}$ essentially serves as an *extractor* for the prover's secret input. As the confidentiality requirements in commitments and ZK proofs are inherently one-sided, these functionalities are natural fits for our transparent enclave execution model. Interestingly, our results show that our model of transparent enclave execution adds to a known list of setup models, including Common Reference Strings [14] or Tamper Proof Hardware Tokens [29], in which *any* multi-party functionality can be securely realized [14, 15].

One subtlety is that a commitment scheme inherently requires *state* to be kept between the commitment and opening phases. We need a way for an enclave to attest to a verifier that it received an input from the prover, and that this same input was later used in a computation. As the prover

can launch multiple enclaves running the same program, the verifier needs to be convinced that the *same* enclave process is used throughout the protocol. This is easily done in our model (and in SGX [26]) by having the hardware platform provide each enclave with a *fresh random nonce*. In order to open a commitment to a different value (i.e., break the binding property), a prover would need to launch two enclaves with the same nonce, or break the integrity or authenticity of the enclave execution.

In Figure 3 we give a protocol $\mathbf{Prot}_{\text{SGP}}$ between a verifier $\mathcal{V}$ and a prover $\mathcal{P}$ which realizes the $\mathcal{F}_{\text{SGP}}$ functionality. The prover runs a "wrapper" around prog in an enclave (Figure 4), that saves $\mathcal{P}$'s committed input, attests to the correct execution of prog and later reveals the input.

THEOREM 1. *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks (EU-CMA). Then $\mathbf{Prot}_{\text{SGP}}$ securely realizes $\mathcal{F}_{\text{SGP}}$ in the $\mathcal{F}_{\text{TEE}}$-hybrid model, for static adversaries.*

PROOF. Our proof is similar to that used for the $\mathcal{F}_{\text{CP}}$ functionality in [15]. Let $\mathcal{A}$ be a static adversary against $\mathbf{Prot}_{\text{SGP}}$in the $\mathcal{F}_{\text{TEE}}$-hybrid model. We construct an ideal-world adversary (the simulator) Sim such that no environment $\mathcal{Z}$ can distinguish between interactions with $\mathcal{A}$ and $\mathbf{Prot}_{\text{SGP}}$ or with Sim and $\mathcal{F}_{\text{SGP}}$.

We consider two cases, where either $\mathcal{P}$ or $\mathcal{V}$ is corrupted (i.e., controlled by the adversary), and prove indistinguishability via a sequence of hybrids. In the following, a probability is considered *negligible* if it is a negligible function of the security parameter λ.

Corrupted Prover: Consider the following sequence of hybrids, starting from a real-world execution of $\mathbf{Prot}_{\text{SGP}}$. Hybrid H_1 lets Sim emulate $\mathcal{F}_{\text{TEE}}$. H_2 rules out forgery attacks on Σ. H_3 ensures that a single enclave is used throughout the protocol and H_4 ensures that the prover correctly forwards information between $\mathcal{F}_{\text{TEE}}$ and $\mathcal{V}$.

Hybrid H_1 proceeds as in the real protocol, except that Sim emulates $\mathcal{F}_{\text{TEE}}$ by generating a keypair $(\mathsf{pk}_{\text{TEE}}, \mathsf{sk}_{\text{TEE}})$ for Σ and publishing pk_{TEE}. When the adversary $\mathcal{A}$ wants to communicate with $\mathcal{F}_{\text{TEE}}$, Sim records $\mathcal{A}$'s messages and faithfully emulates $\mathcal{F}_{\text{TEE}}$'s behavior.

As $\mathcal{A}$'s view in H_1 is emulated perfectly as in the real execution, we have that $\mathcal{Z}$ cannot distinguish between H_1 and the real execution.

Hybrid H_2 is the same as H_1 except for the following modifications. If $\mathcal{A}$ invoked $\mathcal{F}_{\text{TEE}}$ with a correct message $(\mathsf{idx}, \text{"install"}, \widehat{\mathsf{prog}})$, then for every "resume" call to $\widehat{\mathsf{prog}}$, Sim records a tuple (msg, σ) where msg is the output of $\widehat{\mathsf{prog}}$ and σ is the produced attestation under sk_{TEE}. Let Ω denote the set of all such tuples. Whenever $\mathcal{A}$ sends a signature σ to $\mathcal{V}$, let msg be the message over which $\mathcal{V}$ verifies σ in $\mathbf{Prot}_{\text{SGP}}$. Then, if $(\mathsf{msg}, \sigma) \notin \Omega$, Sim aborts.

We can show that H_2 is indistinguishable from H_1 by a reduction to the EU-CMA property of Σ: If $\mathcal{A}$ does not send one of the correct messages in Ω to $\mathcal{V}$, $\mathcal{V}$'s signature verification would fail with all but negligible probability — otherwise, $\mathcal{Z}$ and $\mathcal{A}$ can be leveraged to construct an adversary that succeeds in a signature forgery.

Hybrid H_3 proceeds as in H_2, except for the following difference. When $\mathcal{A}$ sends $(\mathsf{sid}, N, \sigma_{\text{C}})$ to $\mathcal{V}$, let idx be the identifier of the enclave running $\widehat{\mathsf{prog}}$ that produced N and σ_{C} (if no such idx exists, Sim aborts). Then, if $\mathcal{A}$ ever re-invokes "commit" on enclave idx or if $\mathcal{A}$ sends a message to $\mathcal{V}$ containing a signature σ generated by an enclave with a different identifier $\mathsf{idx}' \neq \mathsf{idx}$, Sim aborts.

First note that if an idx as defined above doesn't exist, then $\mathcal{V}$'s signature verification will fail (i.e., the sid or nonce sent by $\mathcal{A}$ do not match the ones signed by $\widehat{\mathsf{prog}}$). As $\mathcal{F}_{\text{TEE}}$ samples new randomness r on every "resume" invocation, launching multiple enclaves running $\widehat{\mathsf{prog}}$ or re-calling "commit" will produce a new nonce $N' \neq N$ with overwhelming probability, implying that $\mathcal{V}$'s signature verification for σ_{P} or σ_{O} will fail. H_3 and H_2 are thus indistinguishable.

Hybrid H_4 proceeds as in H_3 with the following differences. When $\mathcal{A}$ sends $(\mathsf{sid}, N, \sigma_{\text{C}})$ to $\mathcal{V}$, Sim records the input $\mathsf{inp}_{\mathcal{P}}$ that $\mathcal{A}$ sent to $\mathcal{F}_{\text{TEE}}$ to obtain σ_{C}. Sim forwards the verifier's input $\mathsf{inp}_{\mathcal{V}}$ to $\mathcal{A}$. When $\mathcal{A}$ sends $(\mathsf{sid}, \mathsf{outp}, \sigma_{\text{P}})$ to $\mathcal{V}$, Sim aborts if outp and σ_{P} were not obtained from a "resume" call to $\widehat{\mathsf{prog}}$ with input $(\mathsf{sid}, \text{"proof"}, \mathsf{inp}_{\mathcal{V}})$. Sim also aborts if $\mathcal{A}$ sends a message $(\mathsf{sid}, \mathsf{inp}, \sigma_{\text{O}})$ to $\mathcal{V}$ where $\mathsf{inp} \neq \mathsf{inp}_{\mathcal{P}}$.

H_4 and H_3 are trivially indistinguishable, as in both of the described abort cases, $\mathcal{V}$'s signature verification fails.

It remains to argue that H_4 is indistinguishable from the ideal execution: When $\mathcal{A}$ sends $(\mathsf{sid}, N, \sigma_{\text{C}})$ to $\mathcal{V}$, Sim sends the recorded $\mathsf{inp}_{\mathcal{P}}$ to $\mathcal{F}_{\text{SGP}}$. When $\mathcal{V}$ sends $\mathsf{inp}_{\mathcal{V}}$ to $\mathcal{F}_{\text{SGP}}$, Sim forwards it to $\mathcal{A}$. When $\mathcal{A}$ sends $(\mathsf{sid}, \mathsf{outp}, \sigma_{\text{P}})$ to $\mathcal{V}$, Sim sends $(\mathsf{sid}, \text{"ok"})$ to $\mathcal{F}_{\text{SGP}}$. Finally, when $\mathcal{A}$ sends $(\mathsf{sid}, \mathsf{inp}_{\mathcal{P}}, \sigma_{\text{O}})$ to $\mathcal{V}$, Sim can send $(\mathsf{sid}, \text{"open"})$ to $\mathcal{F}_{\text{SGP}}$. Hybrid H_4 is thus equivalent to an ideal world execution.

Corrupted Verifier: Simulating the verifier's view is considerably simpler, as $\mathcal{V}$ only sends a single message in $\mathbf{Prot}_{\text{SGP}}$. We directly describe Sim's behavior:

- Upon receiving a message $(\mathsf{sid}, \text{"commit"})$ from $\mathcal{F}_{\text{SGP}}$, Sim sends a valid message $(\mathsf{sid}, N, \sigma_{\text{C}})$ to $\mathcal{A}$, for a randomly chosen nonce N.
- Sim sends the input $\mathsf{inp}_{\mathcal{V}}$ received from $\mathcal{A}$ to $\mathcal{F}_{\text{SGP}}$ and obtains outp. Sim then creates a message $(\mathsf{sid}, \mathsf{outp}, \sigma_{\text{P}})$ for $\mathcal{A}$, where σ_{P} is a valid signature under sk_{TEE} over $\mathsf{inp}_{\mathcal{V}}$, outp and N.
- Finally, when the commitment to $\mathcal{F}_{\text{SGP}}$ is opened, Sim obtains $\mathsf{inp}_{\mathcal{P}}$ from $\mathcal{F}_{\text{SGP}}$ and can open the commitment to $\mathcal{A}$ by creating a valid σ_{O} over $\mathsf{inp}_{\mathcal{P}}$ and N.

The simulation of $\mathcal{V}$'s view is perfect: $\mathcal{V}$ obtains valid digital signatures $\sigma_{\text{C}}, \sigma_{\text{P}}, \sigma_{\text{O}}$, that are linked by a randomly chosen nonce N. The real-worlds and ideal-world executions are thus indistinguishable to $\mathcal{Z}$. □

For simplicity, the given $\mathcal{F}_{\text{SGP}}$ functionality only captures a single instance of a sealed-glass proof. Extending $\mathcal{F}_{\text{SGP}}$ to multiple instances requires "independence" of commitments and proofs [14]. Non-malleability is immediately guaranteed in our scheme by the security of $\mathcal{F}_{\text{TEE}}$'s signature scheme (i.e., creating a new commitment or proof requires creating an attestation). Another concern are relay attacks: $\mathcal{V}$ could take an attestation received from $\mathcal{P}$ and submit it to another party before $\mathcal{P}$ opens his commitment. In our setting, there is a very simple solution to this problem, already included in $\mathbf{Prot}_{\text{SGP}}$: simply include the prover's identity [1] $\mathcal{P}$ in the

[1]Alternatively, we can link the proof to a single verifier by including $\mathcal{V}$'s identity in the signature.

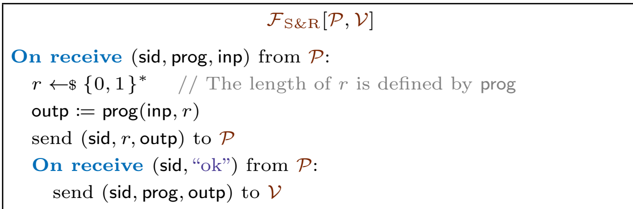


Figure 5: **Ideal Seal-and-Randomize functionality.**

first attestation produced by $\mathcal{F}_{\text{TEE}}$ (further attestations are then also linked to this identity through the nonce N).

Seal and Randomize.

In Section 6, we describe a number of applications of sealed-glass proofs in which the verifier's challenge is simply some random string. For instance, in our motivating example, the verifier's input is an arbitrary "exploit environment". We observe that this scheme can be made *non-interactive*, by having the trusted hardware platform generate randomness on the verifier's behalf. The corresponding ideal "seal and randomize" functionality $\mathcal{F}_{\text{S\&R}}$ is shown in Figure 5.

Using $\mathcal{F}_{\text{TEE}}$ to transform an interactive proof-system into a non-interactive one is reminiscent of the Fiat-Shamir transform for Sigma protocols [21]. An important difference is that $\mathcal{F}_{\text{TEE}}$'s source of randomness cannot be assumed to behave like a random oracle! The reason is that the prover may launch arbitrarily many enclaves running the same program, obtaining fresh randomness for each execution.

In our applications, we work around the prover's ability to adaptively restart executions, by applying standard soundness amplification techniques. More precisely, we can ensure that with all but negligible probability, the verifier obtains a reasonable estimate on the probability $\Pr_r[\mathsf{prog}(\mathsf{inp}_{\mathcal{P}}, r) = \mathsf{true}]$, for r chosen uniformly at random (see Section 7.2).

4. BEYOND TRANSPARENT EXECUTION

Thus far, we have seen how authenticated and integrity protected execution enables efficient constructions of sealed-glass proofs. We show a number of applications of this primitive in Section 5 and Section 6. Notably, combining such proofs with smart contracts can form the basis of knowledge marketplaces with strong fair-exchange guarantees.

The assumption at the heart of this work (namely that enclave execution of arbitrary programs is fully transparent to hosts) is intentionally restrictive. Indeed, our primary goal is to demonstrate the broad usability of trusted hardware that provides only integrity and authenticity. We will see that in a number of applications, though, slight relaxations to this strong assumption may enable more efficient constructions. Here we discuss and motivate two such relaxations to the transparent enclave model, namely *covert-channel resistance* and the ability to *secure cryptographic keys*.

4.1 Covert Channel Resistance

In our setting, a covert channel is a means for the host to feed data to a (maliciously crafted) program running in an enclave, other than through exposed interfaces.[2] For instance, a program in an SGX enclave could use timing primitives to "listen" for signals sent by the host [17, 26].

As enclaves attest to the code they are running, it is reasonable to assume that covert channels would be detected by remote verifiers through code inspection or preconditions on code compilation (e.g., use of a compiler with information-flow controls). Crucially, we note that it is much easier to ensure that no data spuriously *enters* an enclave, than to ensure that no data *leaves* (i.e., leaks from) an enclave. Indeed, the former goal does not necessarily require an absence of side-channels, but simply the inability of the enclave code to listen. For instance, it is easy to check whether the attested code abuses timing functionalities.

For applications such as the bug-bounty system described in Section 7, however, the prover's secret input may itself be viewed as a piece of code called by a *loader program*. (e.g., the input could be a piece of code specifying an exploit.) This secret code cannot be included in the enclave at load-time, as it should not be part of the attestation sent to the verifier. Rather, the prover's code should be loaded *dynamically*, as possible in an upcoming extension to SGX [25].

In such a setting, covert channels become a real concern, as the executed code is not visible to the verifier and cannot be trusted a priori. For instance, the prover could commit to a piece of code that simply consists of a "shell" that cheats by listening on a covert channel to an external process that itself abuses the transparency of the enclave to extract the verifier's challenge! For these applications, we thus require the stronger assumption that either the hardware platform or the loader program can ensure that the prover's dynamically loaded code not communicate with the host. This could be achieved by restricting the programming model for dynamically loaded code (e.g., by removing access to timing functionalities), through use of an approved compiler in an enclave that attests to the validity of its output binary, or through standard sandboxing techniques.

4.2 Confidential Functionalities

The transparent enclave execution model is highly conservative: it assumes complete leakage of a program's state. Here we slightly relax this assumption, and consider enclaves that can be trusted with the confidentiality of select cryptographic primitives. We motivate this "intermediate" model with three observations. First, many cryptographic primitives have known *leakage-resilient* constructions that guarantee security even under partial data leakage. Second, side-channel countermeasures are already routinely used in cryptographic implementations, and incur modest overhead [51]. Third, the security of platforms such as SGX inherently depends on the assumption that the platform's attestation mechanism remains secure. Extending this assumption to other cryptographic primitives appears natural.

Costan and Devadas argue that applying leakage protection solely to cryptographic keys and algorithms is useless if the aim is data confidentiality [17]. However, they fail to note that protecting cryptographic material (and only that) may still have applications for authentication and integrity.

One natural extension to the programming model described in Section 2.1 is the *efficient* setup of an *authenticated channel* to a remote party. If an enclave knows the public key of a remote client, an authenticated channel could in principle be set up using $\mathcal{F}_{\text{TEE}}$: The enclave and remote client could simply digitally sign their messages. Such use of $\mathcal{F}_{\text{TEE}}$ would translate in SGX to use of the quoting enclave

[2]The use of covert channels to pass information *from* the enclave to the host is not considered, as the enclave's execution is already assumed to be transparent.

to sign all messages—an extremely inefficient approach. Instead, we might in practice assume the confidentiality of the key-exchange and MAC primitives in client-side TLS code, given our three observations above. Under this assumption, an enclave could authenticate data from a remote web server over HTTPS, and attest to properties such as the presence or *availability* of data. The recently proposed Town Crier system [50] makes use of this functionality to feed trustworthy data to *smart contracts*, an application we also consider in the context of a fair bug-bounty marketplace (Section 5.2).

5. KNOWLEDGE MONETIZATION

In Section 3, we described how a trusted hardware platform realizing the $\mathcal{F}_{\mathrm{TEE}}$ functionality could be used to create "sealed" zero-knowledge proofs. We now further explore how the $\mathcal{F}_{\mathrm{SGP}}$ and $\mathcal{F}_{\mathrm{S\&R}}$ functionalities can be combined with cryptocurrencies and smart contract systems to create marketplaces for digital knowledge.

For clarity, we will make a slight adjustment to our notation: We consider a Seller $\mathcal{S}$ that possesses some piece of knowledge inp and wishes to sell this information to another party, a Buyer $\mathcal{B}$, for a reward \$Reward. With regard to sealed-glass proofs, the Seller is the *prover*, and the Buyer is the *verifier*. Without loss of generality, we assume the existence of a program prog such that the inputs of interest to the Buyer are those for which $\mathsf{prog}(\mathsf{inp}) = \mathsf{true}$.

While $\mathcal{S}$ may prove possession of inp in zero knowledge without disclosing it to $\mathcal{B}$ (using previously described techniques), a problem arises in a direct exchange of inp for \$Reward between the two parties: The first mover is at a disadvantage. If $\mathcal{S}$ reveals inp to $\mathcal{B}$ before receiving \$Reward, she risks not being paid. If $\mathcal{B}$ pays \$Reward before receiving inp, he risks not obtaining inp. To ensure against such failures and establish a workable marketplace for inp, it is necessary to enforce *fair exchange*. Fair exchange may be achieved by a trusted party that swaps \$Reward and inp honestly between $\mathcal{S}$ and $\mathcal{B}$ (or intervenes when a swap fails [34]).

In this section, we show how sealed-glass proofs may be combined with modern cryptocurrency systems to emulate such a trusted party. We go further, however, and also show that while fair exchange is necessary to achieve a robust marketplace, it is not sufficient. Fair exchange alone does not ensure *fulfillment of a purchase or sale offer*. For example, Buyer $\mathcal{B}$ might offer a bug bounty for a piece of software, causing potential sellers to invest resources in a search for bugs. $\mathcal{B}$ could then subsequently reject any offer she receives from any Seller $\mathcal{S}$. $\mathcal{B}$ can then beneficially learn of the existence of a bug, or cancel her offer if she finds a bug before a Seller does, but $\mathcal{B}$ thereby causes $\mathcal{S}$ to waste resources.

We instead appeal to a stronger notion of honest behavior, fair exchange with *guaranteed fulfillment*. We formally define this notion below, and show how to use SGPs and smart contracts to achieve it. This forms the basis for a *secure marketplace of digital resources*. We further detail one particular application of such a marketplace, a fair and automated bug-bounty system.

5.1 The ZKCP Protocol

In the context of Bitcoin, Maxwell proposed the following simple protocol for fair exchange, referred to as "Zero-Knowledge Contingent Payment" (ZKCP) [33].

Let $\mathsf{prog}(\mathsf{inp}) \to \{\mathsf{true}, \mathsf{false}\}$ be a program that outputs $\mathtt{true}$ iff inp satisfies $\mathcal{B}$'s requirements (e.g., in a bug-bounty system, prog might define an execution environment for a piece of software and inp an exploit). Let H be a cryptographic hash function and Enc a symmetric encryption scheme. Finally, let $\mathsf{hashlock}(h, \mathcal{P}, \$\text{Reward})$ denote a hash-locked transaction, as supported in Bitcoin: On input k such that $h = \mathsf{H}(k)$, the transaction sends \$Reward to party $\mathcal{P}$.

The ZKCP protocol—whose original specification we generalize here as a seller-initiated protocol—is as follows:

1. *Publish predicate:* $\mathcal{B}$ publishes prog, inviting sales offers of satisfying inputs inp.
2. *Create offer:* $\mathcal{S}$ constructs and sends $\mathcal{B}$ a tuple $\mathsf{offer} = (\pi, h, c)$, where π is a non-interactive ZK proof that there exists a bitstring inp such that $\mathsf{prog}(\mathsf{inp}) = \mathtt{true}$, $c = \mathsf{Enc}_{\mathsf{k}}(\mathsf{inp})$, and $h = \mathsf{H}(\mathsf{k})$.
3. *Post conditional transaction:* $\mathcal{B}$ posts transaction $HL = \mathsf{hashlock}(h, \mathcal{S}, \$\text{Reward})$ (possibly with a timeout to claim back \$Reward if the seller aborts).
4. *Claim reward:* $\mathcal{S}$ sends k to HL, obtaining \$Reward.
5. *Recover* inp*:* $\mathcal{B}$ decrypts c using k.

Relationship to Sealed-Glass Proofs.

The ZKCP protocol can be seen as a particular case of a sealed-glass proof system, in which the verifier provides no input. It is actually a special form of a *Commit-and-Prove* functionality [15], wherein the seller provides a commitment to inp and proves something about inp in ZK. [3]

The reasons that the ZKCP makes use of this form of "commitment" to inp are two-fold:

1. The input inp may be large, meaning that it would be impractical and expensive to send it to the blockchain. In the ZKCP protocol, c can be exchanged *off-chain*, so that the opening to the commitment (k, or $\mathsf{k}||r$ as described above) that is sent to the blockchain is small.
2. Bitcoin transactions have limited expressiveness. Payments contingent on more complex operations than a simple hash are not supported.

Restriction (2) can be lifted by using more expressive *smart contracts* [42], but exchanging inp off-chain may still remain preferable for efficiency reasons.

Marketplace Properties Lacking in ZKCP.

The ZKCP protocol is remarkably simple and may suffice for a variety of scenarios of interest. But it fails to achieve three key properties of a practical knowledge marketplace:

1. *Performance:* For many predicates prog of interest, construction of a ZK argument or proof can be prohibitively expensive. Maxwell demonstrated his ZKCP protocol [8,33] using a ZK-SNARK for a trivial application (sale of a Sudoku puzzle solution). Practical knowledge marketplaces will benefit from support for far more complex predicates.
2. *Denial-of-service (DoS) resistance:* In ZKCP, while the hash-lock HL is valid, \$Reward is committed in HL

[3] A minor caveat is that (c, h) is *not* a secure (hiding) commitment to inp in the formal sense: The security of the ZKCP protocol relies on the non-standard assumption that $\mathsf{Enc}_{\mathsf{k}}$ is secure if $\mathsf{H}(\mathsf{k})$ is published. This issue could easily be fixed, while maintaining compatibility with Bitcoin, by setting $h = \mathsf{H}(k||r)$ for a random $r \leftarrow_{\$} \{0,1\}^{\lambda}$, yielding a secure commitment to inp in the random-oracle model.

and unavailable for other purposes. A malicious $\mathcal{S}$ can repeatedly send offers and fail to claim the reward, tying up $\mathcal{B}$'s monetary resources. A workable knowledge marketplace should not permit such abuses.

(3) *Guaranteed fulfillment:* In ZKCP, $\mathcal{B}$ can abort after receiving an offer from $\mathcal{S}$, i.e., fail to execute step 3. As noted above, this diminishes the value of the marketplace, as a potential Seller $\mathcal{S}$ has no guarantee that $\mathcal{B}$ will pay a reward. A knowledge marketplace should instead guarantee fulfillment: $\mathcal{B}$ should pay \$Reward to the first valid seller emerging before a deadline.

Our constructions of ZK proofs using trusted hardware platforms can address limitation (1), which is the most obvious barrier to wide-adoption of a knowledge marketplace. Thus, our first contribution is to show how our sealed-glass proof model can be used to replace expensive cryptographic ZK proofs in the ZKCP protocol by more efficient trusted-hardware-based proofs. We further describe how limitations (2) and (3) can be addressed using smart contracts.

5.2 Secure and Fair Knowledge Marketplaces

We first present the ideal functionality $\mathcal{F}_{\mathrm{KM}}$ that we seek to achieve in a trusted hardware based knowledge marketplace (Figure 6). We define this functionality for a buyer-initiated scheme with multiple sellers; that for a seller-initiated marketplace is analogous.[4]

The $\mathcal{F}_{\mathrm{KM}}$ functionality presented here extends the non-interactive variant of SGPs (i.e., seal-and-randomize proofs) with an additional fair-exchange step. For the sake of clarity, we do not consider the general interactive variant here. To extend SGPs with fair-exchange, $\mathcal{F}_{\mathrm{KM}}$ would require an extra "collection phase", where all sellers commit to their inputs, before the buyer reveals her challenge. $\mathcal{F}_{\mathrm{KM}}$ keeps an internal state st. On transaction initialization, a timer starts in the background and activates after a given timeout.

Observe that while simple, $\mathcal{F}_{\mathrm{KM}}$ accomplishes more than simple fair exchange, achieving all three key properties given above. With regard to DoS resistance, note that in $\mathcal{F}_{\mathrm{KM}}$, the buyer needs sets up a *single* contract and reward, in order to accept offers from all potential sellers. In the ZKCP protocol, each new offer from a seller $\mathcal{S}_i$ would require the buyer to set up a new hash-lock with its own reward.

Smart Contract Realization.

Figure 7 shows a smart contract (as available in Ethereum for instance [11,48]) that leverages the $\mathcal{F}_{\mathrm{S\&R}}$ functionality to realize a buyer-initiated knowledge marketplace. Note that any data sent to a smart contract is inherently *public* (i.e., it is visible on a public blockchain). We thus cannot rely on the smart contract to run prog over the seller's private input.

We first consider that the seller's input is small enough, so that the ciphertext may be sent directly to the smart contract. As we will see, guaranteed fulfillment can then be achieved under no extra assumptions. Later on, we give a more efficient protocol, in which the ciphertext is exchanged off-chain, as in the ZKCP protocol. We discuss different techniques to maintain property (3) in such a setting.

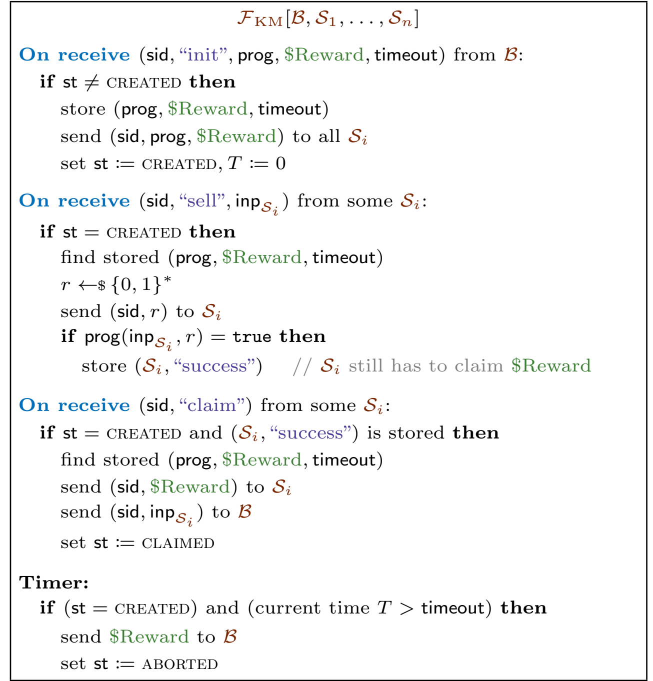


Figure 6: **Ideal buyer-initiated transaction in a knowledge marketplace.**

The KM-Buyer contract first receives a program wrapper description (see Figure 8),[5] a public key, a reward and a timeout value from the buyer. Sellers then have until the timeout to provide valid inputs, using an enclave program (Figure 8) realizing the $\mathcal{F}_{\mathrm{S\&R}}$ functionality from Section 3. The buyer is assured that she can reclaim her reward if no timely seller appears, and the (honest) sellers that one of them will obtain the reward if they provide valid inputs.

THEOREM 2. *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks and that* Enc *is* IND-CPA *secure. Then* KM-Buyer *securely realizes* $\mathcal{F}_{\mathrm{KM}}$ *in the* $\mathcal{F}_{\mathrm{TEE}}$*-hybrid model, for static adversaries.*

Sketch of Proof The proof of this theorem is similar to that of Theorem 1. We describe the main differences below. Note that the timer functionality is the same in the real and ideal worlds and thus need not be simulated.

- As we are considering the $\mathcal{F}_{\mathrm{S\&R}}$ functionality here, the commitment and opening stages of $\mathcal{F}_{\mathrm{SGP}}$ are omitted. The simulator Sim simply records a corrupted seller's input $\mathsf{inp}_{\mathcal{S}_i}$ when $\mathcal{S}_i$ sends it to the program $\widehat{\mathsf{prog}}$ running in $\mathcal{F}_{\mathrm{TEE}}$. Sim then sends $\mathsf{inp}_{\mathcal{S}_i}$ to $\mathcal{F}_{\mathrm{KM}}$ and obtains the randomness r to use in the emulation of $\widehat{\mathsf{prog}}$. As in Theorem 1, a successful distinguisher for the real and ideal worlds implies a forgery attack on Σ.
- Simulating the view of the buyer is again straightforward: when $\mathcal{S}_i$ claims the reward from $\mathcal{F}_{\mathrm{KM}}$, Sim obtains $\mathsf{inp}_{\mathcal{S}_i}$, encrypts it under $\mathsf{pk}_{\mathcal{B}}$, and generates a valid attestation σ to be sent to KM-Buyer.

[4] In a seller-initiated transaction, the seller proves knowledge of an inp such that $\mathsf{prog}(\mathsf{inp}) = \mathsf{true}$, and prog is made public. Potentially multiple buyers may then bid for a reward. After a fixed time, the highest bidder receives inp and the seller receives the winning reward.

[5] We assume that a hash digest is produced when a program is loaded into an enclave, and that this digest is then included in the attestation σ (as in SGX [26]). Sending $\widehat{\mathsf{prog}}$ to a contract means sending this hash digest.

Contract KM-Buyer

On receive (sid, "init", \$Reward, $\widehat{\mathsf{prog}}$, timeout, $\mathsf{pk}_{\mathcal{B}}$) from $\mathcal{B}$:
assert st $= \bot$
assert bank[$\mathcal{B}$] $\geq$ \$Reward
bank[$\mathcal{B}$] := bank[$\mathcal{B}$] $-$ \$Reward
set st := CREATED, $T := 0$

On receive (sid, "sell", c, σ) from some $\mathcal{S}_i$:
assert st = CREATED
assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma, (\widehat{\mathsf{prog}}, (\mathsf{sid}, \mathcal{S}_i, \mathsf{pk}_{\mathcal{B}}, c)))$
bank[$\mathcal{S}_i$] := bank[$\mathcal{S}_i$] + \$Reward
set st := CLAIMED

Timer:
if (st = CREATED) and (current time $T >$ timeout) **then**
bank[$\mathcal{B}$] := bank[$\mathcal{B}$] + \$Reward //no seller found
set st := ABORTED

Figure 7: **Smart contract realizing $\mathcal{F}_{\text{KM}}$.** Messages sent to the contract are stored and are publicly visible.

Enclave program wrapper $\widehat{\mathsf{prog}}$

On input $((\mathsf{sid}, \mathcal{S}_i, \mathsf{inp}_{\mathcal{S}}, \mathsf{pk}_{\mathcal{B}}), r, \mathsf{mem})$:
assert $\mathsf{prog}(\mathsf{inp}_{\mathcal{S}}, r) = \mathsf{true}$
let $c := \mathsf{Enc}_{\mathsf{pk}_{\mathcal{B}}}(\mathsf{inp}_{\mathcal{S}})$
return $(\mathsf{sid}, \mathcal{S}_i, \mathsf{pk}_{\mathcal{B}}, c)$ // include the seller's identity in σ

Figure 8: **Program wrapper used in** KM-Buyer**.**

- Another difference is that we are considering multiple sellers (provers) here. As mentioned in Section 3, binding an attestation to the legitimate seller's identity ensures that attestations are not transferable, i.e., one seller cannot claim the reward for a proof produced by another seller. If an honest seller claims the reward of an honest buyer, the view of a malicious seller can be simulated by replacing the ciphertext c by a random value. As Enc is IND-CPA secure, the seller's view is indistinguishable from a real execution.

□

Implementation Considerations.

The above scheme can in principle be implemented today using trusted hardware platforms and smart contract systems such as SGX [26] and Ethereum [11, 48]. However, some extensions to our threat model and to current smart contracts may be required to achieve the most efficient implementations. We describe some of these below.

Rushing Attacks: For simplicity, Figure 6 omits adversarial delaying and re-ordering of messages sent to a contract. See [30] for a formal modeling of blockchains, and of these attacks in particular. In practice, an adversary $\mathcal{A}$ capable of temporarily delaying messages could launch a *rushing attack*: If $\mathcal{A}$ sees a message (c, σ) before the smart contract acknowledges it, $\mathcal{A}$ could claim it as its own and send it to the smart contract to obtain the reward. Furthermore, if $\mathcal{A}$ corrupts the buyer, it can decrypt c to obtain the seller's input and send this input to the contract to "sell" it to itself.

These attacks are easily prevented by (1) including the seller's identity in the enclave's attestation (see Section 3 and Figure 8); and (2) either having all sellers first commit to their value [27], or even more simply by requiring that the attestation generation process *takes longer than the blockchain's transaction latency*. Delaying attestation generation is easily done in platforms such as SGX, in which enclaves have access to a trusted source of elapsed time [25, 26].

Attestation Verification: Verifying attestations directly in a smart contract may be expensive. Ethereum, for instance, today lacks native support for the proprietary EPID signature scheme used in SGX [9], although it may eventually be added [50]. An alternative approach is to have SGX enclaves directly produce signatures under a private key associated with an Ethereum wallet [50]. In our model, this would require the assumption that enclaves can use a private signing key (other than the platform's main attestation key), without leaking it to the host (see Section 4.2).

Large Inputs: If the seller's input $\mathsf{inp}_{\mathcal{S}}$ is large, sending the ciphertext $c = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{B}}}(\mathsf{inp}_{\mathcal{S}})$ to the blockchain may itself be prohibitively inefficient or monetarily expensive. One way of maintaining guaranteed fulfillment, while allowing some exchanges to occur off-chain as in the ZKCP protocol, is to make use of *trusted data feeds* [11, 48]. The idea is to have a trusted external "notary" attest to the smart contract that the seller followed the protocol honestly.

Concretely, a seller posts her ciphertext (under the buyer's public key) to a *public archive*, and asks a trusted notary (e.g., using Virtual Notary [46] or TLSNotary [45]) to attest to the *availability and validity* of this ciphertext. The notary's attestation could be fed into a smart contract to vouch for the seller's honest behavior and the availability of the ciphertext, assuming data persistence in the archive.

In the relaxed execution model with confidential functionalities explored in Section 4.2, we might instead assume that enclaves can generate and safeguard keys for select cryptographic primitives, such as those used for the handshake and record protocols in TLS . The seller's enclave could then attest directly to the availability and validity of the seller's ciphertext in an HTTPS-enabled public archive (see [50]).

6. APPLICATIONS

In this section, we describe a number of possible applications for the protocols described in Section 3 and Section 5.

6.1 Verifiable Computing

In a *Verifiable Computing* (VC) scheme, one party (usually called the client) outsources a non-confidential computation to another (the server), and receives proof that the computation was performed correctly. This is one of the simplest possible applications of the $\mathcal{F}_{\text{TEE}}$ functionality, and serves as a powerful example of a setting in which execution confidentiality is typically *not* a requirement.[6]

It is easy to see that the sealed-glass proof functionality $\mathcal{F}_{\text{SGP}}$ is a generalization of a VC scheme. Indeed, disregarding the "commit" and "open" interfaces, the "compute" functionality of $\mathcal{F}_{\text{SGP}}$ serves to convince the client of the correct execution of some program prog over both the server's and client's inputs (the server's input may be null).

Although viable VC schemes (i.e., with verification complexity lower than evaluation complexity) can be based on purely cryptographic assumptions [6, 35], trusted-hardware based schemes could be practical today, and for a much larger range of applications. For instance, large-scale *volunteer computing projects* such as SETI@home [1] could use SGX-based VC schemes to efficiently validate computations performed by individual untrusted workers.

[6] Confidential outsourced computing schemes based on SGX have been proposed in [19, 36, 38].

6.2 Zero-Knowledge Proofs

Like VC schemes, Zero-Knowledge (ZK) proofs are a special case of the sealed-glass proof functionality $\mathcal{F}_{\text{SGP}}$. If we omit the input provided by the verifier (and thus the need for a commitment and opening phase), the functionality reduces to a non-interactive proof for the statement $\mathsf{prog}(\mathsf{inp}) = \mathsf{outp}$, where inp is the prover's secret "witness", and the verifier only learns prog and outp.[7]

More formally, if L is a language in $\mathcal{NP}$, then there is an efficient program prog_L such that for any x, there exists a witness y such that $\mathsf{prog}_L(x, y) = \mathsf{true}$ iff $x \in L$. Loading prog_L into an enclave and obtaining an attestation over (prog_L, x) serves as a proof of knowledge of y (the knowledge extractor is simply $\mathcal{F}_{\text{TEE}}$). The ZK property is trivial.

A simple and widely applicable use-case of ZK proofs based on trusted-hardware is in proving knowledge of a hash preimage. As an example, consider the verification of an API key. A user with SGX-enabled hardware can produce an attestation to the fact that she holds an API key k that hashes to some value $h = \mathsf{H}(\mathsf{k})$. The security advantages of this scheme are two-fold: (1) the server only needs to store $\mathsf{H}(\mathsf{k})$ and is thus less vulnerable to key theft in case of a security breach; and (2) the key k required for a successful authentication never leaves the client's machine, and thus can't be intercepted in a network attack.

6.3 Knowledge Marketplaces

We consider two specific applications of the knowledge marketplace functionality presented in Section 5, namely *machine learning competitions* and *bug bounties*. Of course, many other scenarios considered in the literature on ZK proofs and zero-knowledge contingent payments could also be easily instantiated in our model. For instance, examples of ZK proofs for the sale of Sudoku solutions [8], or of factorizations of RSA keys [3] have been proposed.

The applications we consider here are much more general and complex, and it is not immediately evident how to construct traditional ZK proofs for them. While possible in theory [37], the derived proof systems are typically prohibitively inefficient. In contrast, a trusted hardware platform such as SGX can attest to the execution of arbitrary functions, with minimal overhead [38]. As such, the applications described here demonstrate a wide gap between traditional ZK proofs and trusted-hardware-based approaches in their respective practicality and range of applicability today.

Machine Learning (ML) Competitions.

Consider a party that wishes to buy an ML model that performs well for a problem of interest. The party organizes a competition and provides a training set to competitors, while holding out a test set. The competitor showing the best results on this test set exchanges her model for a reward.

In typical ML competitions such as those organized by Kaggle [28], the organizer discloses an *unlabeled* test set, and each participant provides the labels computed by her model. The organizer announces the identity of the winning participant, and exchanges the winning model for the reward. Under Kaggle's terms of service, an organizer must warrant that she will pay a reward to the winning party.

A knowledge marketplace that extends the sealed-glass-proof functionality in Figure 2 enables a ML competition with fair exchange and guaranteed fulfillment. The protocol proceeds in multiple rounds, with appropriate timeouts. The organizer first releases a training set, and all participants commit to a trained model. The organizer then reveals the (labeled) test set and all participants use trusted enclaves to attest to their model's performance. A smart contract can verify each participant's attestation, identify the winner, and fairly exchange the winning model for the reward.

Bug Bounties.

As described in Section 3, a natural motivation for sealed-glass proofs is in the sale of 0-day software exploits. The instantiation of a fair bug-bounty system, which we describe in more detail in Section 7, follows naturally from the knowledge marketplace primitive proposed in Section 5. That system is buyer-initiated, meaning that a buyer solicits exploits.

Although we do not explore them in detail here, seller-initiated transactions are also possible. In such transactions, a seller proposes a testing environment together with an exploit attestation. A smart contract is used to set up an *open ascending price auction* between interested buyers. The seller then encrypts the exploit with the highest bidder's public key and the smart contract fairly exchanges the ciphertext for the highest bid.

7. A FAIR BUG BOUNTY MARKETPLACE

We propose the implementation of a generic bug-bounty marketplace, built on top of SGX [26] and Ethereum [11,48]. We first describe features of the SGX programming model which define the application scope of our system. We then present our system's design and analyze three examples of bug bounties that we designed and constructed.

7.1 The SGX Programming Model

Intel's SGX software development kit lets developers write enclave programs using native C and C++ [26]. A subset of the C and C++ standard libraries are available inside an enclave, with the notable exception of functions that rely on interaction with the (untrusted) host (i.e., system calls). SGX provides special trusted APIs for random number generation, thread synchronization and exception handling. Explicit interaction with the host occurs by means of an "outside call" (OCall) to an untrusted function outside the enclave. Once the function call completes, control is returned to the enclave and the function output is made available.

Because of such constraints, porting arbitrary software into an enclave may prove difficult. One solution would be to make use of a *shielded execution* system such as Haven [5], that aims at running unmodified legacy code in SGX. However, we note that the generality provided by such solutions also comes at a cost, especially in the context of a bug bounty system: As the complexity of the execution environment grows, so does the probability that it itself contains exploitable bugs. In such a case, a seller may be able to convince a buyer of a bug found in an application, when the seller actually exploited a flaw in the execution environment.

In our experiments (see Section 7.3), we focus on demonstrating bug bounties for applications that can be ported to SGX with only minimal modifications. These include "standalone" programs (e.g., many crypto libraries), or modular applications designed to be portable to various embedded systems. For instance, we easily port a SQL library (SQlite)

[7]Note that proofs in $\mathbf{Prot}_{\text{SGP}}$ are non transferable as $\mathcal{F}_{\text{TEE}}$'s attestation includes the prover's identity

and a TLS library (mbedTLS) into SGX enclaves by disabling untrusted OS features (networking, I/O, localization) and mapping others (dynamic memory allocation, randomness generation) to trusted SGX implementations.

7.2 System Design

As some features required to efficiently instantiate the KM-Buyer contract in Figure 7 are not yet available in Ethereum (see Section 5.2), we built a working bug-bounty platform on top of a marketplace that builds upon the original ZKCP protocol, but does not provide all the guarantees of KM-Buyer. Our goal here is to demonstrate a working system that extends the functionality and greatly improves the efficiency of the ZKCP protocol.

More specifically, we forgo the desired guaranteed fulfillment property for now, and consider an interactive protocol in which sellers $\mathcal{S}_i$ use trusted hardware to implement a "seal-and-randomize" proof of exploit. The enclave attests to the correct encryption of this exploit under some symmetric key k, as well as to a commitment *com* to k, both of which are sent off-chain to the buyer. An honest buyer sends $(\mathcal{S}_i, com)$ to a smart contract holding the reward, which $\mathcal{S}_i$ can claim by revealing k. Note that this simplified protocol achieves the *performance* and *DoS resistance* properties we previously mentioned as limitations of the ZKCP protocol.

The Bounty Wrapper.

We built a generic exploit program wrapper similar to the abstraction in Figure 8. This wrapper exposes an interface that the seller uses to attest to the correct execution of her exploit, as well as to the commitment of the symmetric key used to encrypt the exploit. The actual exploit environment is defined in a separate trusted statically-linked library.

The wrapper places no restrictions on the bounty program other than those imposed by the SGX programming model described previously. It is up to the buyer to verify that this program is "sound", meaning that it correctly classifies the seller's exploit as valid or invalid. For instance, the bounty program may interact with the host through OCalls, but neither the SGX execution environment nor the program wrapper can give any guarantees as to the host's responses (or that the host responds at all). The randomness required by the bounty program is made available dynamically through calls to SGX's random number generator.

Soundness Amplification.

In the seal-and-randomize functionality $\mathcal{F}_{\text{S\&R}}$ (Figure 5), the seller provides some input inp and the enclave then computes $\mathsf{prog}(\mathsf{inp}, r)$ where r is a fresh random string (of a length specified by prog). We assume here that the enclave produces an attestation if $\mathsf{prog}(\mathsf{inp}, r) = \mathsf{true}$, and aborts otherwise. Because the seller controls the enclave's execution environment, she may arbitrarily restart this execution and thus force the enclave to sample a new random string.

Assuming the seller is polynomially bounded (a requirement for the security of SGX's attestation mechanism), the seller obtains an attestation with non-negligible probability only if $\Pr_r[\mathsf{prog}(\mathsf{inp}, r) = \mathsf{true}]$ is non-negligible, where r is sampled uniformly at random. Stronger guarantees for the buyer can be obtained using traditional soundness amplification techniques, as described below.

Let $p \coloneqq \Pr_r[\mathsf{prog}(\mathsf{inp}, r) = \mathsf{true}]$. Then, the $\mathcal{F}_{\text{S\&R}}$ functionality can be extended to compute an estimate $\tilde{p}$ of p, by computing $\mathsf{prog}(\mathsf{inp}, \cdot)$ for n *independent* values r. If the seller launches a new enclave running prog, n fresh independent random strings are sampled. By a standard Hoeffding bound [23], the probability that $\tilde{p}$ and p differ by more than a constant additive factor δ is negligible if we choose $n = \Omega(\lambda)$. The buyer thus specifies a target success probability t and an allowed error δ. The enclave then computes $\tilde{p}$ and produces an attestation iff $\tilde{p} \geq t$. The buyer is thus guaranteed that exploits with true success probability lower than $t - \delta$ will be accepted with only negligible probability.

Application	"Glue" LOC	Time (ms)	Proof Size (B)
Wrapper	365	37	1152
SQL Injection	+166	2290[8]	+166
X.509 Validation	+195	42	+220

Table 1: **Complexity and performance of exploit proofs.** Lines of code provided are for the "glue" code required to set up an exploit environment (excluding the vulnerable library) and attest to the result. Timings are for exploit execution and attestation generation. The proof size comprises SGX's attestation as well as the encrypted exploit.

7.3 Bug Bounty Examples

We now present three examples of bug bounties that can be performed in our system. The first two correspond to exploits specified by a piece of *data*, and we have successfully implemented these in the current SGX platform. Table 1 summarizes the code complexity and performance results for these exploit proofs. The third example describes a general framework for proving exploits defined as *code* implementing a MITM attack. We present preliminary evidence of how to securely prove knowledge of such exploits on top of SGX2 [25] under the additional assumption of *covert-channel resistance* as motivated in Section 4.1.

SQL Injections in Login Forms.

We built a simple web login form that takes as input an HTTP Post message, extracts a username and password, and matches these against a user database. The HTTP processing makes use of Facebook's Proxygen library [20] and the underlying database is built on SQLite [40]. Our application is vulnerable to a simple SQL Injection attack [43], as it directly copies the user's credentials into a SQL statement. Although SQL Injections are highly recognizable attacks with simple and efficient countermeasures, they remain one of the primary reported exploits in industry [43].

This setting illustrates the usage of the seal-and-randomize functionality: If the program populates the user database in a pre-determined manner, the seller could simply provide a valid login by choosing one of the entries in the database. The buyer could also hardcode hashed and salted random passwords into the program, but this doesn't fully prevent the seller from running an (expensive) offline attack to recover valid credentials and sell a (worthless) exploit.

In our system, the seller provides her exploit to a running enclave, which *then* generates random user credentials to populate the database with. This ensures that the seller's exploit is chosen *independently* from the random environment in which it will be tested. As the seller's exploit is very small (less than 256 bytes), sending the ciphertext directly to a smart contract appears reasonable.

[8] Timing is for 10,000 independent runs. The estimation error $|p - \tilde{p}|$ is larger than 5% with probability at most 2^{-80}.

Differential Testing of Certificate Validation Logic.

A generic way of proving that an implementation of a specific standard contains a bug is through differential testing: one proves that some input produces conflicting results on different implementations of the same standard. This technique has been successfully applied to the detection of certificate verification bugs in various TLS libraries [10]. By crafting randomly "mutated" X.509 certificates, the authors found that the produced *frankencerts* could reveal serious flaws in the certificate validation logic.

Here, we propose to use the same differential testing technique (for a known malicious certificate) to reveal a bug in a library. The seller loads a program with two different TLS certificate verification modules (we experimented with OpenSSL [44] and mbedTLS [2]) and asks the enclave to attest that one of the modules accepted the certificate while the other rejected it. In the vast majority of cases, this points to a bug in the "accepting" library [10].

One issue we encountered is that SGX enclaves do not have access to a trusted source of (absolute) time, which is required in the certificate validation process (incidentally, many bugs discovered in [10] were related to incorrect verification of a certificate's validity period). However, we noticed that for the differential testing process described here, it is sufficient that the time used by both libraries is *consistent*. We thus allow the seller to provide the "current time" as part of the exploit, and then attest that the two libraries behave differently *for the certificate and time provided by the host*.

As an example of an exploit proof, we consider a bug discovered in [10]: in an earlier version of mbedTLS, certificates would be accepted even if *not yet valid*. We craft an exploit consisting of a standard self-signed X.509 certificate and a "current time" set in the past. The certificate is accepted by the unpatched version of mbedTLS but correctly rejected by OpenSSL, thus proving knowledge of a bug.

MITM Attacks on TLS Handshakes.

Consider a generic adversary that can see, delay, drop or otherwise alter messages sent by two parties over a network. Such man-in-the-middle (MITM) attacks encompass a large range of attacks on HTTPS connections. We consider here a prominent application to TLS handshakes.

To prove knowledge of such exploits, we require a secure and trusted "simulation environment". To this end, we set up an enclave running honest server and client applications (that know each others' certificates) that will perform a TLS handshake over an adversarial network. The goal of the prover is to show her attack is successful in extracting some secret (e.g., a key) from the honest applications.

If we were to assume that the enclave environment guarantees confidentiality, this could easily be achieved: The adversarial network would be simulated by having each message between client and server be mediated by the host (the prover). If the prover can provide a secret at the end of this interaction, the transcript of all messages sent between client and server constitutes a proof of exploit.

In our transparent model, things are a little trickier. As the client's and server's secrets are expected to irremediably leak to the prover through side-channels, we need the prover to *commit* to her exploit code before the secrets are generated. This is achieved by loading the prover's exploit code into a separate enclave. The network is then simulated by forwarding all messages to this "adversary enclave". This will become possible with Intel's upcoming SGX2 processors [25], which will support *dynamic memory management*. One could then use a trusted *loader enclave* to dynamically load and run the prover's exploit code.

For the exploit to be "meaningful", however, we still need to ensure that the "untrusted" exploit code, once loaded, only receives information through well-specified interfaces. In light of our discussion in Section 4.1, the concern could arise that the prover transmits information to her exploit code over covert channels. As we argued, this can be prevented by restricting functionalities available to dynamically loaded code using well-known sandboxing techniques.

8. RELATED WORK

The use of trusted hardware to ensure integrity, authenticity, and confidentiality of computation dates back to platforms such as XOM [31] and Aegis [41]. These secure processors, and their successors such as SGX [26], aim to create secure enclaves isolated from the host's software stack.

Attacks against these platforms have primarily focused on breaking confidentiality, by exploiting data leakage from various side channels [17, 38, 49]. However, to the best of our knowledge, no successful attacks have been demonstrated against execution integrity or authenticity. Previous works either dismiss side-channel attacks as out of scope [5, 19, 36], or suggest more defensive architectures [18, 22, 32]. Our work appears to be the first to explicitly forgo the goal of execution confidentiality, and to explore the range of applications that remain achievable under these weaker assumptions.

In parallel, verifiable computing (whether in ZK or not) has also been an active field of study in the cryptography community. Despite encouraging recent improvements, primitives such as SK-SNARKs [6, 35] or garbled circuits [24] remain impractical for the vast majority of applications.

Finally, a number of recent works have shown how to use cryptocurrencies to incentivize fair behavior in multiparty computations, through threats of monetary penalties [4, 7]. The use of Bitcoin transactions and smart contracts for the sale of information has been considered in [3, 27, 33].

9. CONCLUSION

We have introduced Sealed-Glass Proofs (SGPs), a novel cryptographic functionality based on transparent enclaves trusted for execution authenticity and integrity, but not confidentiality. When realized on top of Intel's SGX, SGPs sidestep many concerns about data leakage in the presence of side-channels. We have shown that SGPs generalize many interesting functionalities in the UC framework, inducing verifiable computing, commitment schemes, and ZK proofs.

We have described how SGPs combined with smart contracts can create knowledge marketplaces with previously unachieved combinations of efficiency, denial-of-service resistance, fair exchange, and fulfillment guarantees. As an application, we have implemented an end-to-end bug-bounty system leveraging Intel's SGX and the Ethereum blockchain. We have used our system to prove knowledge of a SQL-Injection bug in a sample web login form and inconsistencies in certificate validation logic between two TLS libraries.

Finally, we have proposed future paths to even broader and more efficient constructions, either through moderate relaxations to the transparent enclave model, or through extensions to smart contract and trusted hardware platforms.

10. REFERENCES

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An experiment in public-resource computing. *CACM*, 45(11):56–61, Nov. 2002.

[2] ARM Limited. mbedTLS. https://tls.mbed.org/.

[3] W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. https://eprint.iacr.org/2016/451, 2016.

[4] S. Barber, X. Boyen, E. Shi, and E. Uzun. Bitter to better—how to make Bitcoin a better currency. In *FC'12*, pages 399–414. Springer, 2012.

[5] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with Haven. *TOCS*, 33(3):8, 2015.

[6] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. Snarks for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO'13*, pages 90–108. Springer, 2013.

[7] I. Bentov and R. Kumaresan. How to use bitcoin to design fair protocols. In *CRYPTO'14*, pages 421–439. Springer, 2014.

[8] S. Bowe. pay-to-sudoku. https://z.cash/zkcp3.pdf.

[9] E. Brickell and J. Li. Enhanced privacy id from bilinear pairing. https://eprint.iacr.org/2009/095, 2009.

[10] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE SP'14*, pages 114–129. IEEE, 2014.

[11] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform.

[12] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS'01*, pages 136–145. IEEE, 2001.

[13] R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC'07*, pages 61–85. Springer, 2007.

[14] R. Canetti and M. Fischlin. Universally composable commitments. In *CRYPTO'01*, pages 19–40. Springer, 2001.

[15] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *STOC'02*, pages 494–503. ACM, 2002.

[16] R. Canetti and T. Rabin. Universal composition with joint state. In *CRYPTO'03*, pages 265–281. Springer, 2003.

[17] V. Costan and S. Devadas. Intel SGX explained. https://eprint.iacr.org/2016/086, 2016.

[18] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. http://eprint.iacr.org/2015/564, 2015.

[19] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang. M2R: Enabling stronger privacy in MapReduce computation. In *USENIX Security'15*, pages 447–462, 2015.

[20] Facebook. Proxygen. https://github.com/facebook/proxygen.

[21] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO'86*, pages 186–194. Springer, 1986.

[22] C. W. Fletcher, M. v. Dijk, and S. Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC'12*, pages 3–8. ACM, 2012.

[23] W. Hoeffding. Probability inequalities for sums of bounded random variables. *JASA*, 58(301):13–30, 1963.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security'11*, volume 201, 2011.

[25] Intel Corp. Intel® Software Guard Extensions Programming Reference. Reference no. 329298-002, October 2014.

[26] Intel Corp. Intel® Software Guard Extensions Evaluation SDK. https://software.intel.com/en-us/sgx-sdk, 2015.

[27] A. Juels, A. Kosba, and E. Shi. The Ring of Gyges: Investigating the future of criminal smart contracts. https://eprint.iacr.org/2016/358, 2016.

[28] Kaggle. Data science competitions. https://www.kaggle.com.

[29] J. Katz. Universally composable multi-party computation using tamper-proof hardware. In *EUROCRYPT'07*, pages 115–128. Springer, 2007.

[30] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy- preserving smart contracts. In *IEEE SP'16*. IEEE, 2016.

[31] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[32] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS'15*, pages 87–101. ACM, 2015.

[33] G. Maxwell. Zero knowledge contingent payment. https://en.bitcoin.it/wiki/Zero_Knowledge_Contingent_Payment.

[34] S. Micali. Fair electronic exchange with invisible trusted parties. https://people.csail.mit.edu/silvio/CommercialPapers/Fair%20Electronic%20Exchange.pdf.

[35] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE SP'13*, pages 238–252. IEEE, 2013.

[36] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE SP'15*, pages 38–54. IEEE, 2015.

[37] A. Shamir. IP = PSPACE. *JACM*, 39(4):869–877, 1992.

[38] E. Shi, F. Zhang, R. Pass, S. Devadas, and D. Song. Trusted hardware: Life, the composable universe, and everything. Manuscript, 2015.

[39] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing your faults from telling your secrets:

Defenses against pigeonhole attacks. arxiv.org/abs/1506.04832, 2015.
[40] SQLite. https://www.sqlite.org/.
[41] E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *ICS'03*, pages 160–171. ACM, 2003.
[42] N. Szabo. Smart contracts. http://szabo.best.vwh.net/smart.contracts.html, 1994.
[43] The Open Web Application Security Project. Top 10–2013. https://www.owasp.org/index.php/Top_10_2013, 2013.
[44] The OpenSSL Project. OpenSSL. http://www.openssl.org/.
[45] TLSNotary - A mechanism for independently audited https sessions. https://tlsnotary.org/TLSNotary.pdf.
[46] Virtual-Notary - a free and secure electronic attestation service. http://virtual-notary.org/.
[47] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *SIGARCH Comput. Archit. News*, volume 35, pages 494–505. ACM, 2007.
[48] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
[49] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE SP'15*, pages 640–656. IEEE, 2015.
[50] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. https://eprint.iacr.org/2016/168/, 2016.
[51] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. http://eprint.iacr.org/2005/388, 2005.