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 models computation possible in an isolated execution environment with *unbounded leakage*, and thus in the face of arbitrary side-channels. 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 SGPs enable a range of practical applications. Our key observation is that SGPs permit safe verifiable computing in zero-knowledge, as data 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. Our 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 isolated software execution environments that could lead to many practical applications of secure multiparty computation. For instance, Intel's newly released Software Guard Extensions [1] (SGX) let developers 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 [1], the confidentiality goal appears elusive. Recent work shows that enclaves may leak large amounts of sensitive information to a malicious host through their memory access patterns [2], [3]. Additional software or physical side-channels [4]–[8] could further compromise an enclave's secrets. Such attacks are not unique to SGX, and a number of defenses have been developed over the years [6]–[9], usually to protect highly sensitive cryptographic code. It is reasonable to assume that trusted hardware platforms can apply these methods 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 [3], [9]–[13] as the resulting performance degradation is non-negligible [10], [14], [15] 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 achieved in practice. We define a formal model of *transparent enclave execution*, wherein 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 (even in the presence of side channels), but that programs running in enclaves *leak unbounded information* to the host. As such, we aim to minimize the size of the trusted computing base (TCB) that need be assumed immune to side-channel leakage.

The question immediately arises: *Is this weaker model of trusted hardware useful?* Many prior works have proposed using trusted hardware for secure outsourcing of computations, including protection of cryptographic keys (Flicker) [16], Digital Rights Management (Aegis) [14], Map-Reduce jobs (M^2R) [17], data analytics (VC3) [18], and even legacy Windows binaries (Haven) [19]. Yet, these applications critically rely on enclave confidentiality to hide data from an untrusted cloud. Successful side-channel attacks (on Haven) have already been demonstrated [2]. While some works identified the utility of trusted hardware for "secret-less" applications such as verifiable computing [14], [16], our work appears to be the first to explicitly forgo enclave confidentiality, and explore the range of applications that remain achievable under these weaker assumptions.

Our key insight is that transparent enclaves provide suffi-

cient guarantees to securely realize a number of functionalities characterized by a "uni-directional" *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 [20], 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*: 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 scheme can be made non-interactive by leveraging randomness provided by the enclave environment.

We propose a compelling application of SGPs in achieving *fair exchange* of a secret for a monetary reward. Such transactions support a number of use-cases, including 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 [21] 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 [22] 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* [23] 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 leveraging SGX and Ethereum [24], [25]. We use our system to showcase sales of 0-day exploits in several applications: (1) A SQL injection bug in a web login form; (2) Disparities in certificate validation logic in different TLS libraries; and (3) A generic framework for MITM attacks, in particular attacks on TLS handshakes. Our solution is highly efficient: the exploit proofs required the addition of only a few hundred lines of trusted code to the tested applications, and incurred negligible performance overhead compared to running the exploits in a non-isolated environment.

Contributions. 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 and 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 leveraging SGX and Ethereum, and report experiments for sample proofs of 0-day exploits.

2. Preliminaries

In this section, we formalize our model of transparent enclave execution and discuss its relevance in view of known side-channel attacks against trusted hardware platforms.

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 [2]. 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, masking, constant time implementations) [6].

We first introduce some notation used in this work.

$\mathcal{P}, \mathcal{V}$	Host (prover) equipped with trusted hardware, remote verifier
prog	A program to run in an enclave
$\mathsf{inp}, \mathsf{outp}, \mathsf{mem}$	The program's input, output and memory tape
$\mathsf{sid}, \mathsf{idx}$	Session identifier and Enclave identifier
λ, N	Security parameter, cryptographic nonce
Σ, KGen	A Signature Scheme and its key generation algorithm
$\leftarrow_{\$}$	Uniformly random sampling from the right-hand set
$\mathcal{A}, \mathsf{Sim}, \mathcal{Z}$	Real-world adversary, ideal-world adversary (the simulator) and environment in the UC Framework [20]

Formal Model. For our purposes, an enclave is an isolated software container loaded with some program. We modify the trusted hardware abstraction of Shi et al. [10], [26], to account for our relaxed enclave transparency assumption.

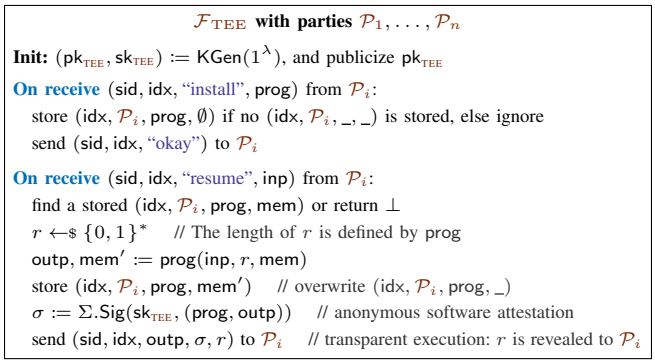

Figure 1: Transparent Enclave Execution functionality.

Our hope is to narrow the gap between formal abstractions and real-world implementations of trusted hardware platforms. We define a functionality $\mathcal{F}_{\text{TEE}}$ (Transparent Enclave Execution) for a trusted hardware platform used by multiple hosts $\mathcal{P}_i$ (Figure 1). $\mathcal{F}_{\text{TEE}}$ abstracts an 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 (or verifier) $\mathcal{V}$. The program gets access to trusted randomness (e.g., as in SGX [1]), but its execution is fully transparent to the host (i.e., all program inputs including the 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 state, e.g., through side channels. This setting could be generalized by considering various constrained information leakage models (see Section 4.2).

The host $\mathcal{P}$ first loads a program prog into an enclave with identifier idx (see Figure 1). On a “resume” call, prog is run on the given input and fresh randomness, to produce some output. All program inputs and outputs are revealed to $\mathcal{P}$. $\mathcal{F}_{\text{TEE}}$'s “resume” call can capture settings where prog is a *stateful* program that handles successive inputs from $\mathcal{P}$ (potentially received from other parties). To realize sealed-glass proofs in Section 3, we will require an enclave to store an input committed by the host. Note that a host can launch multiple enclaves with different idxs running the same program prog. Enclaves generate attestations on each “resume” invocation. An attestation is a digital signature, under a secret key sk_{TEE}, of the loaded program (i.e., a hash digest of the loaded binary) and the program's output outp.

Adversarial Model and Universal Composability. We prove the security of our protocols in the Universal Composability (UC) framework [20], to illustrate the strong security and modularity guarantees offered by our transparent enclave execution model. Specifically, $\mathcal{F}_{\text{TEE}}$ is a *setup assumption* [20], that models the functionality offered by real-world trusted hardware platforms. Therefore, $\mathcal{F}_{\text{TEE}}$ uses a “real” signature scheme Σ, rather than an idealized variant [27], as doing so would correspond to a less natural assumption on the setup provided by trusted hardware.[1] Throughout this work, we assume that Σ is existentially unforgeable under chosen message attacks,[2] and that all parties know the public key pk_{TEE} at the start of a protocol.

Each party $\mathcal{P}$ is identified by a unique id (simply denoted $\mathcal{P}$) and a session id sid, obtained from an *environment* $\mathcal{Z}$ [20]. Parties send messages over *authenticated channels*; an adversary $\mathcal{A}$ observes all messages sent over the network. For ideal functionalities, we use the standard “delayed outputs” terminology [20]: when a functionality $\mathcal{F}$ *sends a public delayed output* msg *to party* $\mathcal{P}_i$, this means that msg is first sent to the ideal-world adversary (the simulator Sim), and forwarded to $\mathcal{P}_i$ only after acknowledgment by Sim.

Note that $\mathcal{F}_{\text{TEE}}$ is a *local* functionality: a host $\mathcal{P}$ can talk to its trusted hardware without leaking this interaction to $\mathcal{A}$. In contrast, when $\mathcal{A}$ corrupts a party, it gets full access to that party's software and hardware. $\mathcal{A}$ can fully observe all programs executed in trusted hardware, but $\mathcal{A}$ cannot tamper with the hardware's integrity and attestation authenticity.

The proposed $\mathcal{F}_{\text{TEE}}$ currently fails to capture composition with protocols that make use of the same *global* trusted hardware functionality, while implicitly sharing state in the form of that platform's attestation key. Stronger composability guarantees could be obtained in the GUC framework [28], by modeling $\mathcal{F}_{\text{TEE}}$ as a single global setup assumption. Extending our results to GUC introduces some technical artifacts, in order to circumvent some impossibility results in [28]. We defer a formal treatment of $\mathcal{F}_{\text{TEE}}$ as a global setup functionality to a forthcoming work [29].

Relation to Intel's SGX. SGX [1] 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. SGX actually makes use of a *group signature scheme* with efficient revocation support [30]. For simplicity, $\mathcal{F}_{\text{TEE}}$ does not capture revocation issues.

The transparent execution model of $\mathcal{F}_{\text{TEE}}$ highlights known vulnerabilities arising from the execution of arbitrary programs in SGX enclaves. Indeed, although SGX aims at protecting against known software and hardware attacks [1], it has been shown that enclaves can leak considerable information through side channels [2], [3], [15]. Our model pessimistically assumes that computations that have not been

1. Other setup functionalities that make internal use of cryptographic primitives can for instance be found in [28].

2. Following [27], we could then replace $\mathcal{F}_{\text{TEE}}$'s signature scheme by an ideal one. Yet, this requires changing the definitions from [27] slightly, as $\mathcal{F}_{\text{TEE}}$ cannot be corrupted. We do not believe that this approach would simplify the exposition or analysis of our protocols, hence we omit it here.

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. We note that these assumptions are consistent with SGX's design, which explicitly leaves protection against page-fault and timing side-channels in enclave applications to developers [1].

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 $\mathcal{P}$. 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. Moreover, in settings where the challenge r is simply a random string, we can leverage the trusted randomness supplied by $\mathcal{F}_{\text{TEE}}$ to render such a scheme *non-interactive*. We describe applications of SGPs in the construction of a *digital knowledge marketplace*, and a fair bug-bounty program in particular.

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 in which the access-control policy is violated (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.

In a number of applications, notably the ones we describe in Section 6, the challenge can simply be viewed as a random string. For instance, in our motivating example, the system-owner's input is an arbitrary "exploit environment." We observe that the above proposed scheme can be made *non-interactive*, by having the trusted hardware platform generate a random challenge on the system-owner's behalf. 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 [31], except that the randomness is provided by trusted hardware instead of a random oracle.

The $\mathcal{F}_{\text{SGP}}$ Functionality. Sealed-Glass Proofs meet the requirements of the above scenario, permitting, e.g., a sound ZK-proof of knowledge of an exploit. Specified in Figure 2, an SGP can be viewed as combining commitment schemes with verifiable computation (in ZK) over some challenge.

A prover first commits to running a known program on her secret input. Then, the verifier may interactively specify a challenge to include in the prover's computation. Alternatively, the challenge can also take the form of fresh random coins provided to the program during its execution. The key is that the challenge 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, combining SGPs with smart contract systems can form the basis of a *digital knowledge marketplace* with strong guarantees for fair-exchange.

Realizing $\mathcal{F}_{\text{SGP}}$ in the $\mathcal{F}_{\text{TEE}}$ Model. $\mathcal{F}_{\text{SGP}}$ can be seen as an extension of the "commit-and-prove" functionality $\mathcal{F}_{\text{CP}}$ in [32]. Compared to $\mathcal{F}_{\text{CP}}$, $\mathcal{F}_{\text{SGP}}$ further includes a *challenge* phase, that aggregates an optional input provided by $\mathcal{V}$, and randomness provided by the trusted hardware.

As shown in [32], $\mathcal{F}_{\text{CP}}$ can be realized given a secure commitment scheme and an ideal ZK functionality. Our

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

// Prover commits to her input
On receive $(\mathsf{sid}, \text{“commit”}, \mathsf{inp}_{\mathcal{P}})$ from $\mathcal{P}$:
  store $(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}} := \bot)$
  send a public delayed output $(\mathsf{sid}, \text{“receipt”})$ to $\mathcal{V}$

// Verifier provides a challenge (optional)
On receive $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$ from $\mathcal{V}$:
  find $(\mathsf{inp}_{\mathcal{P}}, \_)$ stored or halt
  store $(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}})$
  send a public delayed output $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$ to $\mathcal{P}$

// Compute prog on $\mathsf{inp}_{\mathcal{P}}$, $\mathsf{inp}_{\mathcal{V}}$ (optional), and trusted randomness
On receive $(\mathsf{sid}, \text{“prove”})$ from $\mathcal{P}$:
  find $(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}})$ stored or halt // $\mathsf{inp}_{\mathcal{V}}$ may be $\bot$
  let $\mathsf{outp} := \mathsf{prog}(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}}, r)$, for $r \leftarrow\$ \{0,1\}^*$
  send $(\mathsf{sid}, \text{“proof”}, \mathsf{outp}, r)$ to $\mathcal{P}$, wait to receive $(\mathsf{sid}, \text{“okay”})$ from $\mathcal{P}$
  send a public delayed output $(\mathsf{sid}, \text{“proof”}, \mathsf{outp})$ to $\mathcal{V}$

// Prover reveals her input
On receive $(\mathsf{sid}, \text{“open”})$ from $\mathcal{P}$:
  find $(\mathsf{inp}_{\mathcal{P}}, \_)$ stored or halt
  send a public delayed output $(\mathsf{sid}, \text{“open”}, \mathsf{inp}_{\mathcal{P}})$ to $\mathcal{V}$

Figure 2: Ideal Sealed-Glass Proof. $\mathcal{F}_{\text{SGP}}$ is parametrized by a program prog and involves two parties $\mathcal{P}, \mathcal{V}$. Activation points are *non-reentrant*: If queried more than once, $\mathcal{F}_{\text{SGP}}$ ignores the request. The length of the random input r is specified by prog.

protocol for realizing $\mathcal{F}_{\text{SGP}}$ makes use of $\mathcal{F}_{\text{TEE}}$ to directly realize both commitments and ZK proofs. Indeed, $\mathcal{F}_{\text{TEE}}$ can be used as an *extractor* for the prover's secret input. As the confidentiality requirements in commitments and ZK proofs are inherently one-sided, these primitives are natural fits for our transparent enclave execution model. Interestingly, our results show that $\mathcal{F}_{\text{TEE}}$ adds to a known list of setups, including Common Reference Strings [33] or Tamper Proof Hardware Tokens [34], from which *any* multi-party functionality can be securely realized [32], [33].

Note that $\mathcal{F}_{\text{SGP}}$ requires *state* to be kept between the commit and open phases (i.e., the committed input has to be stored). This is easily done in the $\mathcal{F}_{\text{TEE}}$ model, by having an enclave store the committed input in memory. Yet, as the prover can launch multiple enclaves running the same program, the verifier must be convinced that the *same* enclave process is being used throughout the protocol. This is also easily achieved in our model (and in SGX [1]), by having the hardware platform provide each enclave with a *fresh random nonce*, N. To open a commitment to a different value (i.e., to break the binding property), a prover must launch two enclaves that generate the same nonce, which happens only with negligible probability.

In Figure 3 we give a protocol $\mathbf{Prot}_{\text{SGP}}$ which realizes the $\mathcal{F}_{\text{SGP}}$ functionality. The prover $\mathcal{P}$ runs a "wrapper" $\widehat{\mathsf{prog}}$ around prog in an enclave (Figure 4): it saves $\mathcal{P}$'s committed input (and binds this input to a nonce N with the attestation σ_{C}), attests to the correct execution of prog (with σ_{P}) and later reveals the input to $\mathcal{V}$ (by producing σ_{O}).

Theorem 1. *Assume that $\mathcal{F}_{\text{TEE}}$'s 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 $\mathcal{F}_{\text{TEE}}$ model, for static adversaries.*

The proof of this theorem is given in an extended version of this work [35].

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

Prover $\mathcal{P}$:

On input $(\mathsf{sid}, \text{“commit”}, \mathsf{inp}_{\mathcal{P}})$ from the environment $\mathcal{Z}$:
  set $\mathsf{idx} := 0$
  send $(\mathsf{sid}, \mathsf{idx}, \text{“install”}, \widehat{\mathsf{prog}})$ to $\mathcal{F}_{\text{TEE}}$, wait to receive $(\mathsf{sid}, 0, \text{“okay”})$
  send $(\mathsf{sid}, \mathsf{idx}, \text{“resume”}, (\text{“commit”}, \mathcal{P}, \mathsf{inp}_{\mathcal{P}}))$ to $\mathcal{F}_{\text{TEE}}$
  wait to receive $(\mathsf{sid}, \mathsf{idx}, (\text{“receipt”}, \mathcal{P}, N), \sigma_{\text{C}}, \_)$ from $\mathcal{F}_{\text{TEE}}$:
  send $(\mathsf{sid}, \text{“receipt”}, N, \sigma_{\text{C}})$ to $\mathcal{V}$

On receive $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$ from $\mathcal{V}$:
  store $\mathsf{inp}_{\mathcal{V}}$ and output $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$

On input $(\mathsf{sid}, \text{“prove”})$ from $\mathcal{Z}$:
  find $\mathsf{inp}_{\mathcal{V}}$ stored or set $\mathsf{inp}_{\mathcal{V}} := \bot$
  send $(\mathsf{sid}, \mathsf{idx}, \text{“resume”}, (\text{“prove”}, \mathsf{inp}_{\mathcal{V}}))$ to $\mathcal{F}_{\text{TEE}}$
  wait to receive $(\mathsf{sid}, \mathsf{idx}, (\text{“proof”}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N), \sigma_{\text{P}}, \_)$ from $\mathcal{F}_{\text{TEE}}$:
  send $(\mathsf{sid}, \text{“proof”}, \mathsf{outp}, \sigma_{\text{P}})$ to $\mathcal{V}$

On input $(\mathsf{sid}, \text{“open”})$ from $\mathcal{Z}$:
  send $(\mathsf{sid}, \mathsf{idx}, \text{“resume”}, \text{“open”})$ to $\mathcal{F}_{\text{TEE}}$
  wait to receive $(\mathsf{sid}, \mathsf{idx}, (\text{“open”}, \mathsf{inp}_{\mathcal{P}}, N), \sigma_{\text{O}}, \_)$ from $\mathcal{F}_{\text{TEE}}$:
  send $(\mathsf{sid}, \text{“open”}, \mathsf{inp}_{\mathcal{P}}, \sigma_{\text{O}})$ to $\mathcal{V}$

Verifier $\mathcal{V}$:

On receive $(\mathsf{sid}, \text{“receipt”}, N, \sigma_{\text{C}})$ from $\mathcal{P}$:
  assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{C}}, (\widehat{\mathsf{prog}}, (\text{“receipt”}, \mathcal{P}, N)))$, output $(\mathsf{sid}, \text{“receipt”})$

On input $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$ from $\mathcal{Z}$:
  store $\mathsf{inp}_{\mathcal{V}}$ and send $(\mathsf{sid}, \text{“challenge”}, \mathsf{inp}_{\mathcal{V}})$ to $\mathcal{P}$

On receive $(\mathsf{sid}, \text{“proof”}, \mathsf{outp}, \sigma_{\text{P}})$ from $\mathcal{P}$:
  find $\mathsf{inp}_{\mathcal{V}}$ stored or set $\mathsf{inp}_{\mathcal{V}} := \bot$
  assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{P}}, (\widehat{\mathsf{prog}}, (\text{“proof”}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N)))$
  output $(\mathsf{sid}, \text{“proof”}, \mathsf{outp})$

On receive $(\mathsf{sid}, \text{“open”}, \mathsf{inp}_{\mathcal{P}}, \sigma_{\text{O}})$ from $\mathcal{P}$:
  assert $\Sigma.\mathsf{Vf}(\mathsf{pk}_{\text{TEE}}, \sigma_{\text{O}}, (\widehat{\mathsf{prog}}, (\text{“open”}, \mathsf{inp}_{\mathcal{P}}, N)))$
  output $(\mathsf{sid}, \text{“open”}, \mathsf{inp}_{\mathcal{P}})$

Figure 3: Protocol realizing $\mathcal{F}_{\text{SGP}}$. Both parties' activation points are non-reentrant, i.e., the corresponding actions are executed at most once.

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

On input $((\text{“commit”}, \mathcal{P}, \mathsf{inp}_{\mathcal{P}}), r, \mathsf{mem})$:
  parse r as a nonce N of length λ //store $\mathsf{mem}' = \mathsf{inp}_{\mathcal{P}} || N$
  return $(\text{“receipt”}, \mathcal{P}, N)$ //outputs will be included in σ_{C}

On input $((\text{“prove”}, \mathsf{inp}_{\mathcal{V}}), r, \mathsf{mem})$:
  parse mem as $\mathsf{inp}_{\mathcal{P}} || N$
  let $\mathsf{outp} := \mathsf{prog}(\mathsf{inp}_{\mathcal{P}}, \mathsf{inp}_{\mathcal{V}}, r)$ //store $\mathsf{mem}' = \mathsf{inp}_{\mathcal{P}} || N$
  return $(\text{“proof”}, \mathsf{outp}, \mathsf{inp}_{\mathcal{V}}, N)$ //outputs will be included in σ_{P}

On input $(\text{“open”}, \_, \mathsf{mem})$:
  parse mem as $\mathsf{inp}_{\mathcal{P}} || N$
  return $(\text{“open”}, \mathsf{inp}_{\mathcal{P}}, N)$ //outputs will be included in σ_{O}

Figure 4: Program wrapper used in $\mathbf{Prot}_{\text{SGP}}$. Function calls are non-reentrant, i.e., they return $\bot$ if re-invoked.

Realizing Multiple SGPs. For simplicity of exposition, $\mathcal{F}_{\text{SGP}}$ captures a single instance of a sealed-glass proof between two parties. In Section 5, however, we will design protocols that make use of multiple SGPs between different pairs of parties. Security can then be proved in a *hybrid model* [20], in which parties can directly call $\mathcal{F}_{\text{SGP}}$. By direct application of the UC theorem [20], we would need to assume that each SGP is realized using an independent copy of $\mathcal{F}_{\text{TEE}}$ (i.e., with its own unique secret key). This does not correspond to the situation arising in practice, where all copies of a trusted hardware platform (from the same

vendor) share the same attestation key [1], [15].

As noted in Section 2.1, one approach is to prove security in the GUC framework [28], by viewing $\mathcal{F}_{\text{TEE}}$ as a global setup shared by all parties and protocols. However, for our purposes, it will suffice to show that we can realize multiple $\mathcal{F}_{\text{SGP}}$ instances *using a single copy of* $\mathcal{F}_{\text{TEE}}$. Note that $\mathcal{F}_{\text{TEE}}$ allows a party to create an arbitrary number of enclaves (with different identifiers idx). Thus, we consider simply "re-using" $\mathbf{Prot}_{\text{SGP}}$ (with different idx values) for every instance of $\mathcal{F}_{\text{SGP}}$ to be realized between any pair of parties. We claim that this construction is actually secure: First, note that extending $\mathcal{F}_{\text{SGP}}$ to multiple instances requires "independence" of commitments and proofs [33] (i.e., a corrupted party should not be able to "copy" a commitment or proof issued by an honest party in a different SGP instance). We observe that $\mathbf{Prot}_{\text{SGP}}$ already provides these guarantees. Indeed, non-malleability of commitments and proofs is immediately guaranteed by the security of $\mathcal{F}_{\text{TEE}}$'s signature scheme (i.e., creating a new commitment or proof requires creating a valid attestation). Furthermore, each honestly produced attestation in $\mathbf{Prot}_{\text{SGP}}$ is linked to the identity $\mathcal{P}$ of the honest prover (the identity $\mathcal{P}$ is directly included in the first attestation, and further attestations are linked to this identity through the unique nonce N). Thus, these attestations cannot be transferred.

We summarize this discussion by the following claim:

Claim 2. *Let π be a protocol in the $\mathcal{F}_{\text{SGP}}$-hybrid model, that makes multiple calls to $\mathcal{F}_{\text{SGP}}[\mathsf{prog}, \mathcal{P}_i, \mathcal{P}_j]$ for pairs of parties $\mathcal{P}_i, \mathcal{P}_j$. Let $\pi^{\mathbf{Prot}_{\text{SGP}}}$ be the protocol obtained by replacing each instance of $\mathcal{F}_{\text{SGP}}$ in π by an instance of $\mathbf{Prot}_{\text{SGP}}$. Then, $\pi^{\mathbf{Prot}_{\text{SGP}}}$ securely realizes π in the $\mathcal{F}_{\text{TEE}}$ model, and makes use of a single copy of $\mathcal{F}_{\text{TEE}}$.*

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 pass data to a maliciously crafted enclave program, other than through exposed interfaces.[3] For instance, programs in SGX enclaves could abuse timing primitives to "listen" for signals sent by the host [1], [15]. Our assumed abstraction $\mathcal{F}_{\text{TEE}}$ does not allow for such message-passing between hosts and enclave programs. Below, we discuss how covert-channel resistance can be effectively realized in practice.

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.

Yet, for applications such as the bug-bounty system described in Section 7, 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. The prover's code must thus be loaded *dynamically*, as possible in an upcoming extension to SGX [36].

In this setting, covert channels are 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 simple piece of "shell" code that cheats by listening on a covert channel to another process that itself abuses enclave transparency to extract the verifier's challenge! For these applications, we require the stronger assumption that either the hardware platform or the loader program can ensure that the prover's dynamically loaded code cannot communicate with the host. This could be achieved by imposing restrictions on dynamically loaded code (e.g., disabling timing functionalities), or through standard sandboxing techniques.

4.2. Securing Cryptographic Keys

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 two observations. First, many side-channel countermeasures (e.g., constant-time constructions, blinding or masking [6], [7]) are routinely used in the implementation of cryptographic primitives, and incur modest overhead [6]. Second, the security of platforms such as SGX inherently relies on the assumption that the platform's attestation key remains protected.[4] Extending this assumption to cryptographic primitives used in application code seems natural.

3. Covert channels that pass information *from* the enclave to the host are irrelevant, as the enclave's execution is already assumed to be transparent.

4. SGX's documentation mentions techniques to protect cryptographic code against side channels [1]. It is reasonable to assume that side-channel defenses are used in the quoting enclave, to protect the attestation key.

Costan and Devadas argue that applying leakage protection solely to cryptographic keys and algorithms is useless if the aim is data confidentiality [15]. However, we note that protecting cryptographic material (and only that) may still have applications for authentication and integrity.

A simple use-case of trusted hardware is in *efficiently* setting up 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 digitally sign all their messages. Such use of $\mathcal{F}_{\text{TEE}}$ would translate in SGX to use of the quoting enclave 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 two 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 [26] 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 $\mathcal{F}_{\text{TEE}}$ could be used to create "sealed" zero-knowledge (ZK) proofs. We now further explore how SGPs can be extended 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 $\$\mathsf{Reward}$. 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 ZK without disclosing it to $\mathcal{B}$ (using $\mathcal{F}_{\text{SGP}}$), an issue arises if the two parties perform a direct exchange of inp for $\$\mathsf{Reward}$: The first mover is at a disadvantage. If $\mathcal{S}$ reveals inp before receiving $\$\mathsf{Reward}$, she risks not being paid. If $\mathcal{B}$ pays $\$\mathsf{Reward}$ upfront, she might not obtain 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 $\$\mathsf{Reward}$ and inp between $\mathcal{S}$ and $\mathcal{B}$ (or intervenes if a swap fails [37]).

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 the bug is no longer of value to her, 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) [21].

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 some software and inp an exploit). Let H be a cryptographic hash function and Enc a symmetric encryption scheme. Finally, let $\mathsf{hashlock}(h, \mathcal{S}, \$\mathsf{Reward})$ denote a hash-locked transaction, as supported in Bitcoin: On input k from $\mathcal{S}$ such that $h = \mathsf{H}(k)$, the transaction sends $\$\mathsf{Reward}$ to party $\mathcal{S}$.

The ZKCP protocol—whose original specification we generalize here as a buyer-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 a transaction $\mathsf{HL} = \mathsf{hashlock}(h, \mathcal{S}, \$\mathsf{Reward})$ (possibly with a timeout to claim back $\$\mathsf{Reward}$ if the seller aborts).
(4) *Claim reward:* $\mathcal{S}$ sends k to HL, obtaining $\$\mathsf{Reward}$.
(5) *Recover* inp*:* $\mathcal{B}$ decrypts c using k.

Relationship to Sealed-Glass Proofs. The ZKCP protocol is a special case of a SGP, in which the verifier provides no input. It is actually a form of *Commit-and-Prove* functionality [32]: the seller commits to inp and proves something about inp in ZK.[5] 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.

5. Note 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 known. This is easily fixed by setting $h = \mathsf{H}(k||r)$ for a random $r \leftarrow_{\$} \{0,1\}^{\lambda}$. This yields a Bitcoin-compatible secure commitment in the random-oracle model.

Restriction (2) can be lifted by using more expressive *smart contracts* [23], 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, ZK arguments can be prohibitively expensive to construct. The ZKCP protocol was demonstrated [21], [38] 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 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 a valid seller emerging before a deadline.

Our constructions of ZK proofs using trusted hardware 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 discuss a formal modeling of smart contracts [23], [39]. We then present an ideal "knowledge-marketplace" functionality, and describe a protocol leveraging sealed-glass proofs and smart contracts to realize it.

Modeling of Blockchains and Smart Contracts. We adopt the Hawk framework [39], which proposes a formalization of blockchains and smart contracts in the UC framework, with a threat model adapted to distributed consensus protocols. We present some of the main features here, and refer the reader to [39] for further details:

- **Party identifiers:** Each party $\mathcal{P}_i$ has an associated identity (simply denoted $\mathcal{P}_i$) and can send authenticated messages to a smart contract (e.g., messages are digitally signed by a key associated to a user account[24], [25]).
- **Money:** Money is modeled as a public *ledger*, which associates monetary quantities to parties. Parties can transfer money to each other through smart contracts.
- **Timing:** A per-contract timer is modeled as a discrete variable T that the environment can increment, by having all honest parties send `tick` messages to the smart contract. All parties can query a contract's timer.
- **Message Batches:** Smart contracts are assumed to execute in rounds. A new round starts when the timer T is incremented (modeling the addition of a new block to the chain). The adversary can observe every message (a.k.a transaction) sent to a smart contract in a round. At the end of the round, the adversary selects the *order* in which these messages will be processed by the contract. Thus, messages sent to contracts are guaranteed to be delivered in the same round, but in adversarial order. Our protocols are easily extended to a setting where messages are guaranteed delivery after $k > 1$ rounds.

These assumptions are reflected in the design of ideal functionalities as follows: (1) When an ideal smart contract $\mathcal{C}$ receives a message from a party, $\mathcal{C}$ stores the message and executes the *immediate actions* (see [39]) associated with receipt of this message. Such actions model computations and information leakage that occur "off-chain", as well as the adversary's ability to observe messages sent to the blockchain. (2) At the end of a round, $\mathcal{C}$ applies an adversarial permutation to all messages stored in that round. Then, the *delayed actions* [39] (in gray background in Figure 5) of each message are executed, in the specified order. This models computation that occurs in a real-world smart contract, every time a batch of transactions is included in a new block. (3) The ideal contract also has an associated timer, incremented by the environment through honest parties.

In the Hawk framework [39], these generic functionalities are handled by *wrappers* around ideal smart contracts, real-world contracts, and protocols. For simplicity of exposition, we will keep these wrappers implicit in our proposed solution, and refer the reader to [39] for the full details.

Rushing Attacks. The adversary's ability to re-order messages in each round models a broad class of real-world *rushing attacks*: a corrupted party observes a message sent to the blockchain and adaptively crafts its own input to be processed by the contract in the same round.

These attacks have an important impact on contract design. Looking forward, we will build a smart contract allowing multiple sellers to compete for a reward, by submitting a proof that a given ciphertext (under a buyer's public key) holds a valid input. Consider the following attack: A malicious buyer observes a ciphertext sent by a seller to the contract, decrypts it (thus obtaining her coveted input), and then crafts a new message to the contract (acting as a seller) in order to also claim the reward. As the adversary decides on the order in which messages are processed in each round, she can ensure that the buyer's claim is processed first, and thus that she obtains the input for free.

To avoid such attacks, our smart contract will work in two phases: sellers first *commit* to the value of their input, and must wait for a future round in order to claim the reward. Thus, if a malicious buyer intercepts a valid input in round T, she would have to wait until round $T + 1$ to claim the reward, as she must commit first. However, the honest seller's claim will then have been already processed.

$\mathcal{F}_{\text{KM}}[\mathsf{prog}, \mathcal{B}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$ **with adversary** $\mathcal{A}$

// Initialize a buyer-initiated marketplace
On receive $(\mathsf{sid}, \text{“init”}, \$\mathsf{Reward}, \mathsf{timeout})$ from $\mathcal{B}$:
send $(\mathsf{sid}, \text{“init”}, \mathcal{B}, \$\mathsf{Reward}, \mathsf{timeout})$ to $\mathcal{A}$
assert $\mathsf{st} = \bot$; set $\mathsf{st} := \text{CREATED}$, $T := 0$, $D, D_{\mathcal{B}} := \emptyset$
assert $\mathsf{ledger}[\mathcal{B}] \geq \$\mathsf{Reward}$; set $\mathsf{ledger}[\mathcal{B}] := \mathsf{ledger}[\mathcal{B}] - \$\mathsf{Reward}$
$D := \{(\text{“init”}, \mathcal{B}, \$\mathsf{Reward}, \mathsf{timeout})\}$

// Let sellers commit to an input
On receive $(\mathsf{sid}, \text{“commit”}, \mathsf{inp}_{\mathcal{S}_i})$ from $\mathcal{S}_i$:
assert $\mathsf{st} = \text{CREATED}$ and $\mathcal{S}_i$ has not called “commit” before
send $(\mathsf{sid}, \text{“receipt”}, \mathcal{S}_i)$ to $\mathcal{A}$
store $(\mathcal{S}_i, \mathsf{inp}_{\mathcal{S}_i}, T)$; $D := D \cup \{(\text{“receipt”}, \mathcal{S}_i, T)\}$

// A seller that committed in a previous round can try to claim the reward.
// Immediate off-chain actions: execute prog over the committed input.
// Delayed on-chain actions: exchange input and reward.
On receive $(\mathsf{sid}, \text{“claim”})$ from $\mathcal{S}_i$:
assert $\mathsf{st} = \text{CREATED}$ and $(\mathcal{S}_i, \mathsf{inp}_{\mathcal{S}_i}, T')$ is stored for a $T' < T$
assert $\mathcal{S}_i$ has not called “claim” before
$r \leftarrow\!\$\ \{0,1\}^*$; send (sid, r) to $\mathcal{S}_i$, wait to receive $(\mathsf{sid}, \text{“okay”})$
if $\mathsf{prog}(\mathsf{inp}_{\mathcal{S}_i}, r) = \mathtt{true}$:
store $(\mathcal{S}_i, \text{“success”})$
if $\mathcal{B}$ is corrupted by $\mathcal{A}$: send $(\mathsf{sid}, \text{“claimed”}, \mathcal{S}_i, \mathsf{inp}_{\mathcal{S}_i})$ to $\mathcal{A}$
else : send $(\mathsf{sid}, \text{“claimed”}, \mathcal{S}_i)$ to $\mathcal{A}$
assert $(\mathcal{S}_i, \text{“success”})$ is stored and $\mathsf{st} = \text{CREATED}$
set $\mathsf{st} := \text{CLAIMED}$, $\mathsf{ledger}[\mathcal{S}_i] := \mathsf{ledger}[\mathcal{S}_i] + \$\mathsf{Reward}$
$D := D \cup \{(\text{“claimed”}, \mathcal{S}_i)\}$; $D_{\mathcal{B}} := \{\mathsf{inp}_{\mathcal{S}_i}\}$

// Query current smart-contract state
On receive $(\mathsf{sid}, \text{“get-state”})$ from a party $\mathcal{P}$:
if $\mathcal{P} = \mathcal{B}$: send $(\mathsf{sid}, \mathsf{st}, T, D, D_{\mathcal{B}})$ to $\mathcal{B}$
else : send $(\mathsf{sid}, \mathsf{st}, T, D)$ to $\mathcal{P}$

Timer(T): assert $(\mathsf{st} = \text{CREATED})$ and $(T > \mathsf{timeout})$
set $\mathsf{st} := \text{ABORTED}$, $\mathsf{ledger}[\mathcal{B}] := \mathsf{ledger}[\mathcal{B}] + \$\mathsf{Reward}$

Figure 5: Ideal buyer-initiated transaction in a knowledge marketplace. The adversary selects the order in which delayed actions are processed in each round.

As the adversary can choose the message order in each round, we allow that if multiple sellers (honest or corrupt) provide valid inputs in the same round, the adversary can choose which party gets the reward. The contract could also split the reward between all successful sellers in a round. However, sellers could simply create many “dummy parties” proposing the same exploit, to increase their reward share.

The $\mathcal{F}_{\text{KM}}$ functionality. We now present the ideal functionality $\mathcal{F}_{\text{KM}}$ that we seek to achieve in a trusted hardware based knowledge marketplace (Figure 5). We define this functionality for a buyer-initiated scheme with multiple sellers; that for a seller-initiated marketplace is analogous. In a seller-initiated transaction, the seller proves knowledge of inp such that $\mathsf{prog}(\mathsf{inp}) = \mathsf{true}$. Buyers then bid for a reward. After a fixed time, the highest bidder gets inp and the seller receives the winning reward.

Our goal is to extend SGPs (Figure 2) with an additional fair-exchange step. For our purposes, it will be sufficient to consider the “non-interactive” variant of SGPs, in which the verifier does not explicitly provide a challenge, but the computation may make use of trusted randomness. The fairness guarantees we wish to achieve are (1) the buyer is assured that she can reclaim her reward if no timely seller appears, and (2) the sellers are assured that one of them will obtain the reward if they provide valid inputs.

Contract KM-Buyer

On receive $(\mathsf{sid}, \text{“init”}, \widehat{\mathsf{prog}}, \$\mathsf{Reward}, \mathsf{timeout}, \mathsf{pk}_{\mathcal{B}})$ from $\mathcal{B}$:
assert $\mathsf{st} = \bot$ and $\mathsf{ledger}[\mathcal{B}] \geq \$\mathsf{Reward}$
$\mathsf{ledger}[\mathcal{B}] := \mathsf{ledger}[\mathcal{B}] - \$\mathsf{Reward}$
set $\mathsf{st} := \text{CREATED}$, $T := 0$
$D := \{(\text{“init”}, \mathcal{B}, \$\mathsf{Reward}, \mathsf{timeout})\}$, $D' := \{(\widehat{\mathsf{prog}}, \mathsf{pk}_{\mathcal{B}})\}$

On receive $(\mathcal{S}_i, \text{“receipt”})$ from $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$
assert $\mathsf{st} = \text{CREATED}$ and no message $(\mathcal{S}_i, \text{“receipt”})$ was received before
$D := D \cup \{(\text{“receipt”}, \mathcal{S}_i, T)\}$

On receive $(\mathcal{S}_i, \text{“proof”}, (c, \mathsf{pk}_{\mathcal{B}}))$ from $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$
assert $\mathsf{st} = \text{CREATED}$ and $(\text{“receipt”}, \mathcal{S}_i, T') \in D$ for a $T' < T$
set $\mathsf{st} := \text{CLAIMED}$, $\mathsf{ledger}[\mathcal{S}_i] := \mathsf{ledger}[\mathcal{S}_i] + \$\mathsf{Reward}$
$D := D \cup \{(\text{“claimed”}, \mathcal{S}_i)\}$, $D' := D' \cup \{c\}$

On receive $(\mathsf{sid}, \text{“get-state”})$ from a party $\mathcal{P}$: send $(\mathsf{sid}, \mathsf{st}, T, D, D')$ to $\mathcal{P}$

Timer(T): assert $(\mathsf{st} = \text{CREATED})$ and $(T > \mathsf{timeout})$
set $\mathsf{st} := \text{ABORTED}$, $\mathsf{ledger}[\mathcal{B}] := \mathsf{ledger}[\mathcal{B}] + \$\mathsf{Reward}$

Figure 6: Smart contract for realizing $\mathcal{F}_{\text{KM}}$. $\mathcal{A}$ observes all messages and chooses the processing order in each round.

$\mathbf{Prot}_{\text{KM}}[\mathsf{prog}, \mathcal{B}, \mathcal{S}_1, \ldots, \mathcal{S}_n]$

Buyer $\mathcal{B}$:

On input $(\mathsf{sid}, \text{“init”}, \$\mathsf{Reward}, \mathsf{timeout})$ from $\mathcal{Z}$:
$(\mathsf{pk}_{\mathcal{B}}, \mathsf{sk}_{\mathcal{B}}) := \mathsf{KGen}(1^\lambda)$
send $(\mathsf{sid}, \text{“init”}, \widehat{\mathsf{prog}}, \$\mathsf{Reward}, \mathsf{timeout}, \mathsf{pk}_{\mathcal{B}})$ to KM-Buyer

On input $(\mathsf{sid}, \text{“get-state”})$ from $\mathcal{Z}$:
send $(\mathsf{sid}, \text{“get-state”})$ to KM-Buyer, wait to receive $(\mathsf{sid}, \mathsf{st}, T, D, D')$
if $c \in D'$: $D_{\mathcal{B}} := \{\mathsf{Dec}_{\mathsf{sk}_{\mathcal{B}}}(c)\}$ **else** : $D_{\mathcal{B}} := \emptyset$
output $(\mathsf{sid}, \mathsf{st}, T, D, D_{\mathcal{B}})$

Seller $\mathcal{S}_i$:

On input $(\mathsf{sid}, \text{“commit”}, \mathsf{inp}_{\mathcal{S}_i})$ from $\mathcal{Z}$:
send $(\mathsf{sid}, \text{“get-state”})$ to KM-Buyer, wait to receive $(\mathsf{sid}, \mathsf{st}, T, D, D')$
assert $\mathsf{st} = \text{CREATED}$; parse D' as $\{(\widehat{\mathsf{prog}}, \mathsf{pk}_{\mathcal{B}}\}$
// The seller ID serves as session ID for this Sealed-Glass Proof
send $(\mathcal{S}_i, \text{“commit”}, (\mathsf{inp}_{\mathcal{S}_i}, \mathsf{pk}_{\mathcal{B}}))$ to $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$

On input $(\mathsf{sid}, \text{“claim”})$ from $\mathcal{Z}$:
send $(\mathcal{S}_i, \text{“prove”})$ to $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$
wait to receive $(\mathsf{sid}, \text{“proof”}, \mathsf{outp}, r)$ from $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$:
send $(\mathsf{sid}, \text{“okay”})$ to $\mathcal{F}_{\text{SGP}}[\widehat{\mathsf{prog}}, \mathcal{S}_i, \text{KM-Buyer}]$

On input $(\mathsf{sid}, \text{“get-state”})$ from $\mathcal{Z}$:
send $(\mathsf{sid}, \text{“get-state”})$ to KM-Buyer, wait to receive $(\mathsf{sid}, \mathsf{st}, T, D, D')$
output $(\mathsf{sid}, \mathsf{st}, T, D)$

Figure 7: Protocol realizing $\mathcal{F}_{\text{KM}}$ in the $\mathcal{F}_{\text{SGP}}$-hybrid model. Sellers use the program wrapper from Figure 8 in a SGP with the contract KM-Buyer from Figure 6.

Program wrapper $\widehat{\mathsf{prog}}$ to run in $\mathcal{F}_{\text{SGP}}$

On input $((\mathsf{inp}_{\mathcal{S}}, \mathsf{pk}_{\mathcal{B}}), \bot, r)$:
assert $\mathsf{prog}(\mathsf{inp}_{\mathcal{S}}, r) = \mathsf{true}$
let $c := \mathsf{Enc}_{\mathsf{pk}_{\mathcal{B}}}(\mathsf{inp}_{\mathcal{S}})$ and **return** $(c, \mathsf{pk}_{\mathcal{B}})$

Figure 8: Program wrapper for the SGP used in KM-Buyer.

$\mathcal{F}_{\text{KM}}$ keeps an internal state st. A timer starts on contract initialization and activates after a given timeout. This timer is modeled as in [39]: The ideal contract $\mathcal{F}_{\text{KM}}$ is encapsulated in a “wrapper” that records `tick` messages sent by honest parties. When all honest parties have sent a `tick` (as instructed by $\mathcal{Z}$), the variable T is incremented and the **Timer** procedure of $\mathcal{F}_{\text{KM}}$ is called. In addition to handling timing, this wrapper also handles message batching

and adversarial message ordering (see [39]).

As noted above, the real-world contract requires a commitment round to prevent rushing attacks, and this structure is reflected in $\mathcal{F}_{\text{KM}}$. Any party can query $\mathcal{F}_{\text{KM}}$ to obtain the current internal state st, the timer value T, as well as stored data D. Note that only the Buyer can obtain $D_{\mathcal{B}}$, which eventually contains the valid input from the successful seller.

Observe that while simple, $\mathcal{F}_{\text{KM}}$ accomplishes more than standard fair exchange, achieving all three key properties given above. With regard to DoS resistance, note that in $\mathcal{F}_{\text{KM}}$, the buyer 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 6 shows a smart contract (as available in Ethereum for instance [24], [25]) that leverages the $\mathcal{F}_{\text{SGP}}$ functionality to realize a buyer-initiated knowledge marketplace. Note that data sent to a smart contract is inherently *public* (i.e., it is visible on a public blockchain). We thus cannot rely on a smart contract to run prog over the seller's private input. As only the buyer should eventually learn the input of the seller that claimed the reward (indeed, this input is valuable enough to warrant a reward), we cannot use $\mathcal{F}_{\text{SGP}}$ directly either, as $\mathcal{F}_{\text{SGP}}$ discloses the input to $\mathcal{A}$ in the opening phase. Rather, we will use a SGP between a seller and a smart contract, in order to prove to the contract not only that the seller's input is valid, but also that it has been correctly encrypted under a public key provided by the buyer. The program wrapper (for $\mathcal{F}_{\text{SGP}}$) that provides this abstraction is given in Figure 8. The full protocol realizing $\mathcal{F}_{\text{KM}}$ is described in Figure 7.

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 discuss a more efficient protocol, in which the ciphertext is exchanged off-chain, as in the ZKCP protocol. We suggest different techniques to maintain property (3) in such a setting.

The KM-Buyer contract in Figure 6 first receives a program (i.e., a hash digest of the program binary), a public key, a reward and a timeout value from the buyer. Sellers have until the timeout to provide valid inputs, by engaging in a sealed-glass proof with the smart contract. In practice, the ideal $\mathcal{F}_{\text{SGP}}$ functionalities are replaced by $\mathbf{Prot}_{\text{SGP}}$ protocols, with KM-Buyer in the role of the verifier. The actual program loaded into $\mathcal{F}_{\text{TEE}}$ then consists of the wrapper in Figure 8 embedded into the SGP wrapper in Figure 4.

Again, following [39], the timer functionality, message batching, and adversarial message re-ordering in KM-Buyer are handled by a generic "smart-contract wrapper". For simplicity, we keep this wrapper implicit in our protocols. All data processed by a smart contract is publicly available on a blockchain. This is made explicit in KM-Buyer via the data stores D, D', that any party can obtain by querying the contract. The distinction between D and D' is only made to simplify the analysis, with D corresponding exactly to the data also made available by the ideal functionality $\mathcal{F}_{\text{KM}}$.

Theorem 3. *Assume that the signature scheme Σ is existentially unforgeable under chosen message attacks and that* Enc *is* IND-CPA *secure. Then $\mathbf{Prot}_{\text{KM}}$ securely realizes $\mathcal{F}_{\text{KM}}$ in the $\mathcal{F}_{\text{SGP}}$-hybrid model, for static adversaries.*

The proof of this theorem is given in an extended version of this work [35].

Corollary 4. *$\mathcal{F}_{\text{KM}}$ can be securely realized in the $\mathcal{F}_{\text{TEE}}$ model, using a single copy of $\mathcal{F}_{\text{TEE}}$.*

Proof. This follows immediately from the UC Theorem [20] and Claim 2. □

Implementation Considerations. Although, $\mathcal{F}_{\text{KM}}$ could be implemented today using SGX [1] and Ethereum [24], [25], some extensions to our threat model and to current smart contracts are needed to achieve the most efficient solutions. We describe some implementation details below.

- **Rushing Attacks:** Note that $\mathcal{F}_{\text{SGP}}$ prevents an adversary from copying proofs, even if multiple SGPs are realized from a single copy of $\mathcal{F}_{\text{TEE}}$ (see Claim 2). To recall, this is achieved by having honest sellers link $\mathcal{F}_{\text{TEE}}$'s attestations to their identity. Furthermore, by ensuring that the "commit" and "prove" phases of $\mathcal{F}_{\text{SGP}}$ occur in different timing rounds, KM-Buyer ensures that if a malicious buyer observes a seller's ciphertext, she cannot make use of it to claim the reward in the same round.[6]
- **Attestation Verification:** In practice, the ideal $\mathcal{F}_{\text{SGP}}$ in KM-Buyer is replaced by the $\mathbf{Prot}_{\text{SGP}}$ protocol, with KM-Buyer in the role of the verifier. Thus, KM-Buyer needs to verify $\mathcal{F}_{\text{TEE}}$'s attestations. Verifying attestations in smart contracts may be expensive. Ethereum, for instance, lacks native support for the EPID signature scheme used in SGX [30], although it may eventually be added [26]. An alternative is to have SGX enclaves directly produce signatures under a private key associated with an Ethereum wallet [26]. 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 to the contract could be inefficient or monetarily expensive. To maintain guaranteed fulfillment, while letting some exchanges occur off-chain as in the ZKCP protocol, we could use *trusted data feeds* [24], [25]. 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 to a *public archive*, and a trusted notary (e.g., Virtual Notary [40] or TLSNotary [41]) attests to the *availability and validity* of this ciphertext, assuming data persistence in the archive. The notary's attestation is then fed into a smart contract. If we assume that enclaves can generate

6. We can relax the assumption that messages are guaranteed to be delivered in one round, by introducing an intermediate timeout: If messages are guaranteed to be delivered in at most $k > 1$ rounds, KM-Buyer can accept "receipt" messages until some time $T_1 > k$, and then accept "proof" messages between time T_1 and the $\mathsf{timeout} > T_1 + k$.

and safeguard keys for select cryptographic primitives, such as the handshake and record protocols in TLS (see Section 4.2), a seller could use an enclave to directly attest to the availability and validity of a ciphertext in an HTTPS-enabled public archive (see [26]).

6. Applications

Hereafter, we describe possible applications for our relaxed transparent enclave execution functionality, as well as for the SGP-based protocols in Section 3 and Section 5.

6.1. Verifiable Computing

In a *Verifiable Computing* (VC) scheme, one party outsources a non-confidential computation to another, and receives proof that the computation was performed correctly. This is one of the simplest applications of trusted hardware, and a powerful example of a setting in which execution confidentiality is *not* a requirement. VC schemes based on trusted hardware were previously suggested in [14], [16].

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 "prove" 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 [42], [43], solutions based on trusted hardware could be practical today, for a large range of applications. For instance, *volunteer computing projects* such as SETI@home [44] could use SGX to efficiently verify the computations of untrusted workers.

6.2. Zero-Knowledge Proofs

Like VC schemes, Zero-Knowledge (ZK) proofs are a special case of a sealed-glass proof. If we omit the verifier's challenge and the hardware-provided randomness, (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.

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 for which $\mathsf{prog}_L(x, y) = \mathsf{true}$, 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 ($\mathcal{F}_{\text{TEE}}$ is a simple knowledge extractor). The ZK property is trivial.

A widely applicable use-case of ZK proofs based on trusted-hardware is in proving knowledge of a hash preimage. Consider the validation of an API key: An enclave can attest that a user holds a 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 stores $\mathsf{H}(\mathsf{k})$ and is thus less vulnerable to key theft in case of a security breach; and (2) the authentication key k never leaves the client's machine, and thus cannot be intercepted in a network attack.

6.3. Knowledge Marketplaces

We describe two applications of knowledge marketplaces as presented in Section 5: *machine learning competitions* and *bug bounties*. Of course, many other scenarios considered in the literature on ZK proofs and zero-knowledge contingent payments can also be easily instantiated in our model. For instance, ZK proofs for the sale of Sudoku solutions [38], or of RSA factors [22] 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 [45], 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 [10]. 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 [46], 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.

The knowledge marketplace primitive in Figure 5 enables a ML competition with fair exchange and guaranteed fulfillment. An organizer first releases a training set, and all participants commit to trained models. The organizer then reveals the (labeled) test set and participants attest to their model's performance with trusted enclaves. A smart contract verifies each participant's attestation, identifies the winner, and fairly exchanges the winning model for a reward.

Bug Bounties. As described in Section 3, a natural motivation for SGPs is in the sale of 0-day software exploits. In Section 7, we detail the instantiation of a fair bug-bounty system, which 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 [1] and Ethereum [24],

[25]. 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

The SGX SDK lets developers write enclave programs in native C and C++ [1]. A subset of the C and C++ standard libraries are available inside an enclave, with the exception of functions that require interaction with the untrusted host's OS (i.e., system calls). SGX provides trusted APIs for random number generation, thread synchronization and exception handling. Explicit interaction with the host occurs by means of "outside calls" (OCalls) to untrusted functions outside the enclave. Once a function call completes, control is returned to the enclave, together with the function output.

Because of such constraints, porting arbitrary software into an enclave can be difficult. One solution would be to make use of a *shielded execution* system such as Haven [19], 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 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 ported a SQL library (SQlite) 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 their trusted SGX counterparts.

7.2. System Design

As some features required to efficiently instantiate the KM-Buyer contract (Figure 6) are not yet available in Ethereum (see Section 5.2), we built a working bug-bounty platform on top of a marketplace that extends the original ZKCP protocol, but does not provide all the guarantees of $\mathcal{F}_{\text{KM}}$. More specifically, because of the difficulty to verify SGX attestations in smart contracts, we forgo the guaranteed fulfillment property for now, and consider a protocol in which some exchanges occur *off-chain*. Our goal here is to demonstrate a working system that extends the functionality and greatly improves the efficiency of the ZKCP protocol.

In more detail, a seller $\mathcal{S}_i$ uses an enclave to attest to the correct encryption of her exploit under a 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 then claim by revealing k. Note that this protocol achieves the *performance* and *DoS resistance* properties we previously mentioned as limitations of the ZKCP protocol. However, a dishonest buyer could abort after receiving the encrypted exploit, thus denying the reward to the successful seller (the buyer also forfeits her ability to decrypt the exploit).

Probabilistic Exploits. As discussed in Section 3, the randomness provided by the trusted hardware in a SGP can serve as a random challenge for certain exploit proofs. Yet, some exploits may works only with probability $p < 1$. If the seller is "unlucky", a new SGP would have to be instantiated in order to convince the buyer that the exploit works. If such a probabilistic exploit is of value to the buyer, we can make use of standard *tail bounds* to ensure that the buyer obtains a close estimate for the exploits' true success probability.

We assume that the prover provides an exploit inp ($\mathcal{V}$ provides no challenge) and the enclave checks whether $\mathsf{prog}(\mathsf{inp}, r) = \mathsf{true}$. Let $p \coloneqq \Pr_r[\mathsf{prog}(\mathsf{inp}, r) = \mathsf{true}]$ be the success probability of the exploit, for uniformly random r. Then, we can extend the $\mathcal{F}_{\text{SGP}}$ functionality to compute an estimate $\tilde{p}$ of p, by running $\mathsf{prog}(\mathsf{inp}, \cdot)$ for n *independent* random strings. By a Hoeffding bound [47], the probability that $|p - \tilde{p}| > \delta$, for some constant δ, is negligible in the security parameter λ if $n = \Omega(\lambda)$. The buyer thus specifies a target probability t and an allowed error δ. The enclave computes $\tilde{p}$ and produces an attestation iff $\tilde{p} \geq t$. The buyer is guaranteed that exploits with success probability below $t - \delta$ will be accepted with negligible probability.

The Bounty Wrapper. We built a generic exploit wrapper similar to the abstractions in Figure 4 and Figure 8. It exposes an API that a seller uses to attest to the correctness of her exploit, and to a commitment to the key used to encrypt the exploit. The exploit setup is defined in a separate trusted statically-linked library. The wrapper allows for proofs of probabilistic exploits using the above methods. Any randomness required by the exploit program is made available through calls to SGX's random number generator.

We place no restrictions on the bounty program other than those imposed by the SGX programming model. It is up to the buyer to ensure that the program is "sound", i.e., it correctly classifies exploits as valid or invalid.

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.

We implemented the bounty wrapper for probabilistic exploits described above, using Intel's SGX-SDK [1]. We ran experiments on a laptop with a SGX-equipped processor (Intel Core i7-6500U, Dual-Core 2.50GHz, 16GB of RAM). Our code was written in C and C++, and will be released upon publication. Table 1 gives the code complexity and performance results for our exploit proofs. Attesting the validity of these exploits in an enclave incurred no noticeable performance overhead compared to the cost of running the

Application	"Glue" LOC	Time (ms)	Proof Size (B)
Wrapper	365	37	1152
SQL Injection	+166	2290[7]	+166
X.509 Validation	+195	42	+220

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

exploits in non-isolated environments. The prover's performance in our solution is thus essentially optimal. Porting the exploited applications (which have over 100,000 lines of code [48]–[50]) to the SGX programming model required the addition of only a few hundred lines of "glue code". In contrast, a system like Haven [19], capable of running arbitrary applications in SGX enclaves, is comprised of millions of line of code [19]. A complex exploit setup makes it harder for a buyer to be convinced of its soundness, i.e., that it correctly characterizes exploits over the targeted application. We could not measure the time required to verify exploit proofs, as Intel has yet to launch its "Attestation Verification Service" [15]. However, we note that verifying a proof essentially consists in verifying a single digital signature, over a digest of the program binary and outputs (an encryption of the exploit using AES-128, and a hash-based commitment to the secret symmetric key).

Our third example of a bug bounty 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 [36] 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 extracts a username and password from an HTTP Post message, and matches these against a user database. The HTTP processing makes use of Facebook's Proxygen [51] and the database is built on SQLite [50]. Our application is vulnerable to a simple SQL Injection attack [52], as it directly copies the user's credentials into a SQL statement. Although SQL Injections are highly recognizable, and simple and efficient countermeasures exist, they remain one of the primary reported exploits in industry [52].

This attack illustrates the usage of the non-interactive SGP 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, a seller provides her exploit to an enclave, which *then* generates random user credentials to populate the database with. Thus, the seller's exploit is *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 is reasonable.

Differential Testing of Certificate Validation Logic. A generic way of finding bugs in the implementation of a specific standard is via differential testing: one shows that some input produces conflicting results on various implementations of the standard. This method has been used to find certificate verification bugs in TLS libraries [53]. By crafting randomly "mutated" X.509 certificates, the authors revealed serious flaws in libraries' certificate validation logic.

We propose to use the same differential testing technique (for a known malicious certificate) to *"prove"* that a bug is present in a library. The seller loads a program with two different TLS certificate validation modules (we experimented with OpenSSL [48] and mbedTLS [49]) 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 [53].

Note that SGX enclaves do not have access to a trusted source of (absolute) time, which is required for certificate validation (incidentally, many bugs discovered in [53] were due to incorrect verification of a certificate's validity period). Yet, for the differential testing process described here, it is sufficient for the time used by both libraries to be *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 [53]: 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 network adversary that can observe, delay, drop or alter messages sent between parties. Such man-in-the-middle (MITM) attacks encompass a wide range of attacks on HTTPS connections, and TLS handshakes in particular.

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), and have them 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 assume that enclaves guarantee confidentiality, this could easily be achieved: The 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

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

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 [36], which will support *dynamic memory management*. One could then use a trusted *loader enclave* to dynamically load and run the prover's exploit code.

Yet, for the exploit to be "meaningful" we still require that the exploit code, once loaded, only receives information through well-specified interfaces. Recalling our discussion in Section 4.1, a concern is that the prover transmits information to her exploit code over covert channels; this threat can be mitigated through sandboxing or by limiting the functionality of dynamically loaded code.

8. Related Work

Isolated Execution Environments. The use of trusted hardware for secure computation dates back to platforms such as XOM [54], Aegis [14] or TPM-based solutions such as Flicker [16]. These secure processors, and their successors such as SGX [1], aim to create execution environments which provide execution integrity, authenticity and confidentiality. A number of proposed applications leverage these platforms to securely outsource computations. Recent examples built on SGX include M^2R [17], VC3 [18] and Haven [19]. These systems critically rely on enclave confidentiality to protect data and code from an untrusted cloud.

Trusted Hardware and Side Channels. Many attacks on applications built on top of trusted hardware have focused on breaking confidentiality, by exploiting data leakage from side channels, e.g., page-faults [2], [3] or cache timing [15]. To the best of our knowledge, no successful attacks are known against enclave integrity or authenticity, at least for Intel's SGX platform. Many prior works leveraging SGX dismiss side-channel attacks as out of scope [17]–[19], [26]. Others suggest more defensive architectures, with the goal of eliminating or reducing data leakage from enclaves [3], [9], [11]–[13]. Our work appears to be the first to explicitly forgo enclave confidentiality, and explore applications that are achievable under these weaker assumptions.

Verifiable Computing and ZK Proofs. Verifiable computing (possibly in ZK) is an active field of study in the cryptography community. Despite recent improvements, solutions based on ZK-SNARKs [42], [43] or garbled circuits [55], [56] remain impractical in most settings. In the context of a bug-bounty platform, another limitation of these approaches is that they typically require code to be compiled into some specialized circuit representation, which tend to cover only a small subset of programs and bugs of interest.

Bug Bounties. Netscape launched the first bug bounty program in 1995 [57]. Today, bug bounties are widespread in industry, with a fast growth in rewarded discoveries and number of bounty hunters [58]. Illegal markets for 0-day exploits are also emerging, under the cover of anonymity provided by Tor or Bitcoin [59]. These programs and markets do not guarantee *fair-exchange*, as buyers or sellers can abort transactions after obtaining their share of the goods.

While our SGX-based bug bounty platform requires applications to be explicitly ported to the enclave programming model, this results in a significantly smaller exploit setup, compared to "universal" platforms such as Haven [19].

Cryptocurrencies and Fair-Exchange. A number of works have used cryptocurrencies to incentivize fair behavior in multiparty computations, through threats of monetary penalties [60]–[62]. The use of Bitcoin transactions and smart contracts for the sale of information has been considered in [21], [22], [63]. Our work extends such techniques with the use of transparent trusted hardware.

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.

References

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

[2] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P'15*. IEEE, 2015, pp. 640–656.

[3] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets: Defenses against pigeonhole attacks," 2015, arxiv.org/abs/1506.04832.

[4] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *EUROCRYPT'09*. Springer, 2009, pp. 443–461.

[5] P.-A. Fouque, N. Guillermin, D. Leresteux, M. Tibouchi, and J.-C. Zapalowicz, "Attacking RSA–CRT signatures with faults on Montgomery multiplication," in *CHES'12*. Springer, 2012, p. 16.

[6] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing." 2005, http://eprint.iacr.org/2005/388.

[7] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO'96*. Springer, 1996, pp. 104–113.

[8] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *SIGARCH Comput. Archit. News*, vol. 35. ACM, 2007, pp. 494–505.

[9] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Sec'16*, 2016.

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

[11] 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*. ACM, 2015, pp. 87–101.

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

[13] A. Rane, C. Lin, and M. Tiwari, "Raccoon: closing digital side-channels through obfuscated execution," in *USENIX Sec'15*, 2015, pp. 431–446.

[14] E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *ICS'03*. ACM, 2003, pp. 160–171.

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

[16] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *OSR*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[17] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M^{2}R: Enabling stronger privacy in MapReduce computation," in *USENIX Sec'15*, 2015, pp. 447–462.

[18] 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 *S&P'15*. IEEE, 2015, pp. 38–54.

[19] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," *TOCS*, vol. 33, no. 3, p. 8, 2015.

[20] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS'01*. IEEE, 2001, pp. 136–145.

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

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

[23] N. Szabo, "Smart contracts," http://szabo.best.vwh.net/smart.contracts.html, 1994.

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

[25] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.

[26] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *CCS'16*. ACM, 2016.

[27] R. Canetti, "Universally composable signature, certification, and authentication," in *CSFW'04*. IEEE, 2004, pp. 219–233.

[28] R. Canetti, Y. Dodis, R. Pass, and S. Walfish, "Universally composable security with global setup," in *TCC'07*. Springer, 2007, pp. 61–85.

[29] Anonymous Submission, "Formal abstractions for realistic secure processors," 2016, manuscript.

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

[31] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO'86*. Springer, 1986, pp. 186–194.

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

[33] R. Canetti and M. Fischlin, "Universally composable commitments," in *CRYPTO'01*. Springer, 2001, pp. 19–40.

[34] J. Katz, "Universally composable multi-party computation using tamper-proof hardware," in *EUROCRYPT'07*. Springer, 2007, pp. 115–128.

[35] Anonymous Submission, "Sealed-Glass Proofs: Using transparent enclaves to prove and sell knowledge," http://anonymouspaper.com/sgp/sgp.pdf, 2016.

[36] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Intel® Software Guard Extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave," in *HASP'16*. ACM, 2016.

[37] S. Micali, "Simple and fast optimistic protocols for fair electronic exchange," in *PODC'03*. ACM, 2003, pp. 12–19.

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

[39] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *S&P'16*. IEEE, 2016.

[40] "Virtual-Notary," http://virtual-notary.org/.

[41] "TLSNotary," https://tlsnotary.org/TLSNotary.pdf.

[42] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *S&P'13*. IEEE, 2013, pp. 238–252.

[43] 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*. Springer, 2013, pp. 90–108.

[44] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: An experiment in public-resource computing," *CACM*, vol. 45, no. 11, pp. 56–61, Nov. 2002.

[45] A. Shamir, "IP = PSPACE," *JACM*, vol. 39, no. 4, pp. 869–877, 1992.

[46] Kaggle, "Data science competitions," https://www.kaggle.com.

[47] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *JASA*, vol. 58, no. 301, pp. 13–30, 1963.

[48] The OpenSSL Project, "OpenSSL," http://www.openssl.org/.

[49] ARM Limited, "mbedTLS," https://tls.mbed.org/.

[50] "SQLite," https://www.sqlite.org/.

[51] Facebook, "Proxygen," https://github.com/facebook/proxygen.

[52] The Open Web Application Security Project, "Top 10–2013," 2013, https://www.owasp.org/index.php/Top_10_2013.

[53] 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 *S&P'14*. IEEE, 2014, pp. 114–129.

[54] 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*, vol. 35, no. 11, pp. 168–177, 2000.

[55] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *S&P'15*. IEEE, 2015, pp. 359–376.

[56] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar, "TinyGarble: Highly compressed and scalable sequential garbled circuits," in *S&P'15*. IEEE, 2015, pp. 411–428.

[57] The Free Library, "Netscape announces 'Netscape bugs bounty' with release of Netscape Navigator 2.0 Beta." *PR Newswire*, 1995.

[58] Bugcrowd, "The state of bug bounty," https://bugcrowd.com, 2015.

[59] A. Greenberg, "New dark-web market is selling zero-day exploits to hackers," https://www.wired.com/2015/04/therealdeal-zero-day-exploits/, 2015.

[60] I. Bentov and R. Kumaresan, "How to use Bitcoin to design fair protocols," in *CRYPTO'14*. Springer, 2014, pp. 421–439.

[61] S. Barber, X. Boyen, E. Shi, and E. Uzun, "Bitter to better - how to make Bitcoin a better currency," in *FC'12*. Springer, 2012, pp. 399–414.

[62] R. Kumaresan and I. Bentov, "How to use Bitcoin to incentivize correct computations," in *CCS'14*. ACM, 2014, pp. 30–41.

[63] A. Juels, A. Kosba, and E. Shi, "The Ring of Gyges: Investigating the future of criminal smart contracts," in *CCS'16*. ACM, 2016.