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

Florian Tramèr, Fan Zhang, Huang Lin, Jean-Pierre Hubaux, Ari Juels, Elaine Shi

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 (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]–[15] as the resulting performance degradation is non-negligible [10], [16], [17] 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) [18], Digital Rights Management (Aegis) [16], Map-Reduce jobs (M2R) [15], [19], data analytics (VC3) [20], machine learning [14], and even legacy Windows binaries (Haven) [21]. 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 [16], [18], [22], 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
sufficient guarantees to securely realize functionalities characterized by a “uni-directional” resource asymmetry between untrusted parties, e.g., when one party 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 realize a number of interesting primitives in the UC framework [23], 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 [24] 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 [25] 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 [26] 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 [27], [28]. 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. The code used for our experiments is available at https://github.com/framer/sealed-glass-proofs.

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 described hereafter serves as a relaxation of the standard model of trusted hardware (e.g., as in SGX), in which enclave execution is assumed to be fully concealed from the host. This “standard” model of trusted execution has been formalized for instance in [10], [20], [22], [29].

In contrast, 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 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 [6].

We first introduce some notation used in this work.

| $P$, $V$ | Host (prover) equipped with trusted hardware, remote verifier |
| prog | A program to run in an enclave |
| inp, outp, mem | The program’s input, output and memory tape |
| sid, idx | Session identifier and Enclave identifier |
| $\lambda$, $N$ | Security parameter, cryptographic nonce |
| $\Sigma$, $KGen$ | A Signature Scheme and its key generation algorithm |
| $\longleftrightarrow$ | Uniformly random sampling from the right-hand set |
| $A$, Sim, $\mathcal{Z}$ | Real-world adversary, ideal-world adversary (the simulator) and environment in the UC Framework [23] |

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], [29], to
account for our relaxed enclave transparency assumption \(^1\). 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 \( P_i \) (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 \( P \).

In our context, a host (or prover) \( P \) wants to prove statements about the execution of some program \( prog \), to a remote client (or verifier) \( V \). 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 \( P \)). Only the platform's attestation mechanism (more specifically a secret signing key \( sk_{\text{TEE}} \)) 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 \( P \) 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 \( P \). \( \mathcal{F} \text{-TEE} \)’s “resume” call can capture settings where \( prog \) is a stateful program that handles successive inputs from \( P \) (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 \( idx \)s running the same program \( prog \). Enclaves generate attestations on each “resume” invocation. An attestation is a digital signature, under a secret key \( sk_{\text{TEE}} \) 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 [23], to illustrate the strong security and modularity guarantees offered by our transparent enclave execution model. Specifically, \( \mathcal{F} \text{-TEE} \) is a setup assumption [23], that models the functionality offered by real-world trusted hardware platforms. Therefore, \( \mathcal{F} \text{-TEE} \) uses a “real” signature scheme \( \Sigma \), rather than an idealized variant [30], as doing so would correspond to a less natural assumption on the setup provided by trusted hardware. \(^2\)

1. Barbosa et al. [22] use an alternative model to the one of Shi et al. [10], [29] (both assume fully confidential enclave execution). A main technical difference is that the model from Barbosa et al. is game-based (i.e., security properties are defined as adversarial games), whereas the model from Shi et al. and ours is simulation-based (i.e., all security properties are defined as an “ideal functionality” to be emulated). For primitives that require a large number of security properties (e.g., the protocols considered in Section 5 that combine trusted hardware and smart contracts), we believe it is conceptually simpler to express all security properties through a single ideal functionality, rather than through multiple security games. In addition, the simulation paradigm gives strong security guarantees under protocol composition [23], which game-based approaches usually do not.

2. Other setup functionalities that make internal use of cryptographic primitives can for instance be found in [31].
our model allows us to guarantee that carefully designed protocols remain secure even when composed with other protocols (including several instances of the same protocol), that share the same setup $\mathcal{F}_{\text{TEE}}$ (see Claim 2). However, we provide no guarantees when protocols are composed with other (potentially maliciously crafted) protocols that make use of the same $\mathcal{F}_{\text{TEE}}$ setup (which requires the stronger GUC modeling from [32]).

**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 [33]. 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], [17]. Our model pessimistically assumes that computations that have not been explicitly hardened against known side-channel attacks leak their complete execution trace to the host. We do rely on assumptions of execution integrity and secure attestation, however. As we shall see, these two properties actually suffice in a number of interesting applications. 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 prog(inp) = 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.”
The protocols [34], except that the randomness is provided by the trusted hardware platform. The verifier obtains the output of the program, and the prover may later share her secret with verifiable computation (in ZK) over some challenge. Contract systems can form the basis of a multi-party function call that aggregates an optional input provided during its execution. 

Our protocol for realizing the computation, and the prover may later share her secret with the verifier by opening her commitment.

As we show in Section 5, combining SGP with smart contract systems can form the basis of a digital knowledge marketplace with strong guarantees for fairness-exchange. The random coins provided to the program during its execution.

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 SGP with smart contract systems can form the basis of a digital knowledge marketplace with strong guarantees for fairness-exchange.

Realizing SGP in the TEE Model. SGP can be seen as an extension of the “commit-and-prove” functionality of CP in [35]. Compared to CP, SGP further includes a challenge phase, that aggregates an optional input provided by the verifier, and randomness provided by the trusted hardware.

As shown in [35], CP can be realized given a secure commitment scheme and an ideal ZK functionality. Our protocol for realizing SGP makes use of TEE to directly realize both commitments and ZK proofs. 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 TEE adds to a known list of setups, including Common Reference Strings [36] or Tamper Proof Hardware Tokens [37], from which any multi-party functionality can be securely realized [35], [36].

Note that 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 TEE model, by having an enclave store the committed input in memory. Yet, as the prover can launch multiple enclaves running the}

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

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. This is reminiscent of the Fiat-Shamir transform for Sigma protocols [34], except that the randomness is provided by trusted hardware instead of a random oracle.

Figure 3: Protocol realizing SGP. All activation points are non-reentrant, i.e., the corresponding actions are executed at most once.

Figure 4: Program wrapper used in ProtSGP. Function calls are non-reentrant, i.e., they return ⊥ if re-invoked. The random strings r consumed by prog are sampled by TEE (see Figure 1).
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 $\text{Prot}_{SGP}$ which realizes the $F_{SGP}$ functionality. The prover $P$ runs a “wrapper” $\text{prog}$ around $\text{prog}$ in an enclave (Figure 4): it saves $P$’s committed input (and binds this input to a nonce $N$ with the attestation $\sigma_x$), attests to the correct execution of $\text{prog}$ (with $\sigma_y$) and later reveals the input to $V$ (by producing $\sigma_0$).

**Theorem 1.** Assume that $F_{TEE}$’s signature scheme $\Sigma$ is existentially unforgeable under chosen message attacks (EU-CMA). Then $\text{Prot}_{SGP}$ securely realizes $F_{SGP}$ in the $F_{TEE}$ model, for static adversaries.

The proof of this theorem is given in Appendix A

### Realizing Multiple SGP’s

To simplify exposition, $F_{SGP}$ captures a single instance of a sealed-glass proof between two parties. In Section 5, however, we will design protocols that use multiple SGP’s between different pairs of parties. As we have proved that $\text{Prot}_{SGP}$ securely realizes $F_{SGP}$, the UC theorem [23] lets us analyze the security of additional protocols in an hybrid model, in which sub-calls to $\text{Prot}_{SGP}$ are replaced by interactions with $F_{SGP}$. This modularity simplifies the security analysis, as $F_{SGP}$ explicitly defines the information that is leaked to the adversary.

However, for the UC theorem to apply, we have to assume that each copy of $\text{Prot}_{SGP}$ uses an independent copy of the setup $F_{TEE}$ (i.e., with its own unique secret key) [23]. 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], [17]. We would thus like to show that multiple SGP instances can be securely realized using the same underlying copy of $F_{TEE}$.

Our setting here is comparable to that of Canetti and Fischlin [36] in the context of UC-secure commitments in the Common Reference String (CRS) model: They first show how to build a secure (single-use) commitment from a CRS, and then propose a more complex protocol that can be used to realize multiple commitments using the same CRS.

Hereafter, we argue that multiple SGP’s can actually be securely realized by simply re-using $\text{Prot}_{SGP}$ multiple times, with the same $F_{TEE}$. More specifically, note that $F_{TEE}$ allows a party to create an arbitrary number of enclaves (with different identifiers idx). Thus, we consider “re-using” $\text{Prot}_{SGP}$ (with different idx values) for every instance of $F_{SGP}$ to be realized between any pair of parties. We claim that this construction is secure.

As noted in Section 2.1, an alternative solution would be to prove security in the GUC framework [31], by viewing $F_{TEE}$ as a global setup shared by all protocols. Pass et al. [32] recently showed that many functionalities are surprisingly tricky to realize in this model, requiring either additional assumptions or more complex protocols. In this work, we consider a slightly weaker notion of composability, which allows for simpler constructions. Specifically, in our model, composition holds if protocols either do not share the same $F_{TEE}$, or if the protocols are carefully designed such that attestations produced by $F_{TEE}$ in one run of a protocol cannot affect the security of other protocols running concurrently. We refer the reader to the discussion of the UC framework in Section 2.1 for more details.

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

The proof of this claim is mostly similar to the proof of Theorem 1. Hereafter, we review the main differences.

As mentioned above, the challenge is to show that attestations produced by $F_{TEE}$ in one run of $\text{Prot}_{SGP}$ cannot affect the security of other instances of $\text{Prot}_{SGP}$. Consider the case of a corrupt prover $P$ (the case of an honest prover is the same as in the proof of Theorem 1). Because multiple copies of $\text{Prot}_{SGP}$ are now being run with the same underlying $F_{TEE}$, the prover can obtain valid signatures under $sk_{TEE}$ in two ways: either by interacting with $F_{TEE}$ or by observing messages sent by honest parties in other instances of $\text{Prot}_{SGP}$. We show that this ability to observe attestations produced in different runs of the protocol does not aid $P$ in breaking the security of $F_{SGP}$.

First, note that by the unforgeability of $F_{TEE}$’s signature scheme, $P$ cannot get a verifier $V$ to accept an attestation $\sigma$ that was not produced by $F_{TEE}$. Thus, if $V$ accepts a $\sigma$ sent by $P$, then $\sigma$ must come either from an interaction between $P$ and $F_{TEE}$, or from an interaction between some honest party $P’$ and $F_{TEE}$ in a different run of the protocol. However, note that when an honest prover runs $\text{Prot}_{SGP}$, the first attestation $\sigma_c$ produced by $F_{TEE}$ includes the prover’s identity. Further attestations $\sigma_x$ and $\sigma_y$ produced by $F_{TEE}$ are implicitly linked to the same identity by means of the random nonce $N$. Thus, if the corrupt prover $P$ attempts to re-use an attestation $\sigma$ produced by another honest prover $P’$, the verifier $V$ will reject $\sigma$ with overwhelming probability. In summary, when $P$ is corrupt, any attestation accepted by $V$ must come from an interaction between $P$ and $F_{TEE}$, implying that the simulator can extract $P$’s input.

### 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. For instance, an SGX enclave could abuse timing primitives to “listen” for signals sent by the host [1], [17]. Our assumed abstraction $F_{\text{TEE}}$ does not allow for such message-passing between hosts and enclave programs, and our results thus hold under the implicit assumption that no covert communication between host and enclaves occurs. 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 we will describe 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 may 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 [39].

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. Extending this assumption to cryptographic primitives used in application code seems natural.

Costan and Devadas argue that applying leakage protection solely to cryptographic keys and algorithms is useless if the aim is data confidentiality [17]. However, 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 $F_{\text{TEE}}$: The enclave and remote client could digitally sign all their messages. Such use of $F_{\text{TEE}}$ 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 [29] 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 $F_{\text{TEE}}$ 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 $S$ that possesses some piece of knowledge inp and wishes to sell this information to another party, a Buyer $B$, for a reward $\$\text{Reward}$. 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 $\text{prog}(\text{inp}) = \text{true}$.

While $S$ may prove possession of inp in ZK without disclosing it to $B$ (using $F_{\text{SGP}}$), an issue arises if the two parties perform a direct exchange of inp for $\$\text{Reward}$: The first mover is at a disadvantage. If $S$ reveals inp before receiving $\$\text{Reward}$, she risks not being paid. If $B$ pays 4. Covert channels that pass information from the enclave to the host are irrelevant, as the enclave’s execution is already assumed to be transparent.
$\textbf{Reward}$ upfront, she might not obtain $\text{inp}$. To ensure against such failures and establish a workable marketplace for $\text{inp}$, it is necessary to enforce $\text{fair exchange}$. Fair exchange may be achieved by a trusted party that swaps $\text{Reward}$ and $\text{inp}$ between $S$ and $B$ (or intervenes if a swap fails [40]).

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 $\text{fulfillment of a purchase or sale offer}$. For example, Buyer $B$ might offer a bug bounty for a piece of software, causing potential sellers to invest resources in a search for bugs. $B$ could then subsequently reject any offer she receives from any Seller $S$. $B$ 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 $B$ thereby causes $S$ to waste resources.

We instead appeal to a stronger notion of honest behavior, fair exchange with $\text{guaranteed fulfillment}$. We formally define this notion below, and show how to use SGPs and smart contracts to achieve it. This forms the basis for a $\text{secure marketplace of digital resources}$. 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) [24].

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

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

1. $\text{Publish predicate: } B$ publishes $\text{prog}$, inviting sales offers of satisfying inputs $\text{inp}$.
2. $\text{Create offer: } S$ constructs and sends $B$ a tuple offer $= (\pi, h, c)$, where $\pi$ is a non-interactive ZK proof that there exists a bitstring $\text{inp}$ such that $\text{prog}(\text{inp}) = \text{true}$, $c = \text{Enc}_k(\text{inp})$, and $h = H(k)$.
3. $\text{Post conditional transaction: } B$ posts a transaction $\text{HL} = \text{hashlock}(h, S, \text{Reward})$ (possibly with a timeout to claim $\text{Reward}$ if the seller aborts).
4. $\text{Claim reward: } S$ sends $k$ to $\text{HL}$, obtaining $\text{Reward}$.
5. $\text{Recover inp: } B$ 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 $\text{Commit-and-Prove}$ functionality [35]: the seller commits to $\text{inp}$ and proves something about $\text{inp}$ in ZK. The reasons that the ZKCP makes use of this form of “commitment” to $\text{inp}$ are two-fold:

1. The input $\text{inp}$ 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 $k' \mid r$ as described above) is sent to the blockchain is small.
2. Bitcoin transactions have limited expressiveness. Payments contingent on more complex operations than a simple hash are not supported.

Restriction (2) can be lifted by using more expressive smart contracts [26], 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. $\text{Performance:}$ For many predicates $\text{prog}$ of interest, ZK arguments can be prohibitively expensive to construct. The ZKCP protocol was demonstrated [24], [41] 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. $\text{Denial-of-service (DoS) resistance:}$ In ZKCP, while the hash-lock $\text{HL}$ is valid, $\text{Reward}$ is committed in HL and unavailable for other purposes. A malicious $S$ can repeatedly send offers and fail to claim the reward, tying up $B$’s monetary resources. A workable knowledge marketplace should not permit such abuses.
3. $\text{Guaranteed fulfillment:}$ In ZKCP, $B$ can abort after receiving an offer from $S$, i.e., fail to execute step 3. As noted above, this diminishes the value of the marketplace, as a potential Seller $S$ has no guarantee that $B$ will pay a reward. A knowledge marketplace should instead guarantee fulfillment: $B$ should pay $\text{Reward}$ 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 [26], [42]. We then present an ideal “knowledge-marketplace” functionality, and describe a protocol leveraging sealed-glass proofs and smart contracts to realize it.

---

6. Note that $(c, h)$ is not a secure (hiding) commitment to $\text{inp}$ in the formal sense: The security of the ZKCP protocol relies on the non-standard assumption that $\text{Enc}_k$ is secure if $H(k)$ is known. This is easily fixed by setting $h = H(k | r)$ for a random $r \leftrightarrow \{0, 1\}^\lambda$. This yields a Bitcoin-compatible secure commitment in the random-oracle model.
Modeling of Blockchains and Smart Contracts. We adopt the Hawk framework [42], 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 [42] for further details:

- **Party identifiers:** Each party $P_i$ has an associated identity (simply denoted $P_i$) and can send authenticated messages to a smart contract (e.g., messages are digitally signed by a key associated to a user account[27], [28]).

- **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.

These assumptions are reflected in the design of ideal functionalities as follows: (1) When an ideal smart contract $C$ receives a message from a party, $C$ stores the message and executes the immediate actions (see [42]) 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, $C$ applies an adversarial permutation to all messages stored in that round. Then, the delayed actions [42] (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 [42], 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 [42] 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.

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 $F_{KM}$ functionality. We now present the ideal functionality $F_{KM}$ 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
In a seller-initiated transaction, the seller proves knowledge of the key, a reward and a timeout value from the buyer. Sellers can query \( F_{KM} \) to obtain the current internal state \( st \) and the timer value \( T \), as well as stored data \( D \). Note that only the Buyer can obtain \( DB \), which eventually contains the valid input from the successful seller.

Observe that while simple, \( F_{KM} \) accomplishes more than standard fair exchange, achieving all three key properties given above. With regard to DoS resistance, note that in \( F_{KM} \), 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 \( S_i \) 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 [27], [28]) that leverages the \( FSGP \) 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 \( FSGP \) directly either, as \( FSGP \) discloses the input to \( A \) 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 \( FSGP \)) that provides this abstraction is given in Figure 8. The full protocol realizing \( F_{KM} \) is described in Figure 7.

We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.

The KM-Buyer contract in Figure 6 first receives a new offer from a seller \( S_i \) and a commitment from the buyer \( B \) to the offer. We first consider that the seller’s input is small, so that the ciphertext can be efficiently processed by the smart contract. Guaranteed fulfillment is then achieved under no extra assumptions. In Section 7.2, we discuss different techniques to maintain property (3) in a setting where large ciphertexts are exchanged off-chain, as in the ZKCP protocol.
program loaded into $F_{TEE}$ then consists of the wrapper in Figure 8 embedded into the SGP wrapper in Figure 4.

Again, following [42], 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 $F_{KM}$.

To see that our protocol protects against rushing attacks, first note that $F_{SGP}$ prevents an adversary from copying proofs, even if multiple SGPs are realized from a single copy of $F_{TEE}$ (see Claim 2). To recall, this is achieved by having honest sellers link $F_{TEE}$’s attestations to their identity. Furthermore, as the “commit” and “prove” phases of $F_{SGP}$ 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.

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

The proof of this theorem is given in Appendix B.

**Corollary 4.** $F_{KM}$ can be securely realized in the $F_{TEE}$ model, using a single copy of $F_{TEE}$.

**Proof.** Follows from the UC Theorem [23] and Claim 2. \qed

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, a 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 required. VC schemes based on trusted hardware were previously suggested in [16], [18], [22].

It is easy to see that the sealed-glass proof functionality $F_{SGP}$ is a generalization of a VC scheme. Indeed, disregarding the “commit” and “open” interfaces, the “prove” functionality of $F_{SGP}$ 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 [43], [44], solutions based on trusted hardware could be practical today, for a large range of applications. For instance, volunteer computing projects such as SETI@home [45] 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. To see this, consider the ideal ZK functionality $F_{ZK}$ [35], shown in Figure 9. It is parametrized by a relation $R$ for some language $L$ in NP, such that for any $x$, there exists a polynomial-length witness $w$ for which $R(x, w) = 1$, iff $x \in L$.

Now, if we omit the verifier’s challenge and the hardware-provided randomness in $F_{SGP}$, (and thus the need for a commitment and opening phase), the functionality reduces to taking an input inp from $\mathcal{P}$ and sending outp := prog(inp) to $\mathcal{V}$. If we define prog to be a program that on input inp := $(x, w)$, outputs $x$ if $R(x, w) = 1$, we obtain the ZK functionality in Figure 9. Informally, the (computational) soundness of the ZK proof follows from the hardness of forging an attestation from $F_{TEE}$. The zero-knowledge property is trivially satisfied as the verifier only obtains a digital signature over prog and $x$.

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 $h = H(k)$. The security advantages of this scheme are two-fold: (1) the server only stores $H(k)$ 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 [41], or of RSA factors [25] 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 [46], 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 [47], 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 envelopes. 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 SGP 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 verifies each participant’s attestation, identifies the winner, and fairly exchanges the winning model for a reward.

7. A Fair Bug Bounty Marketplace

We propose the implementation of a generic bug-bounty marketplace, built on top of SGX [1] and Ethereum [27], [28]. 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 [21], 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 7.4), 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. Implementing the KM-Buyer Smart Contract

We describe some implementation details and decisions for instantiating the KM-Buyer contract (Figure 6) using Ethereum [27], [28] and SGX [1]. Although KM-Buyer could in principle be instantiated on top of these two technologies today, some extensions to our threat model and to current smart contracts are needed to achieve the most efficient solutions. We describe these limitations and alternative constructions below.

Attestation Verification. In practice, the sub-calls to $F_{SGP}$ made by Prot$_{KM}$ and KM-Buyer (see Figures 6-7) would be replaced by instances of the Prot$_{SGP}$ protocol, with KM-Buyer in the role of the verifier. Thus, the smart contract KM-Buyer needs the ability to verify $F_{TEE}$’s attestations. However, verifying attestations in smart contracts may be expensive. Ethereum, for instance, lacks native support for the EPID signature scheme used in SGX [33], although it may eventually be added [29]. An alternative is to have SGX enclaves directly produce signatures under a private key associated with an Ethereum wallet [29]. In our model, this requires 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). An alternative (described below for dealing with large inputs) is to allow the ciphertext exchange and attestation verification to occur “off-chain”, possibly at the expense of the guaranteed fulfillment property.

Handling Large Inputs. If the seller’s input inp$_S$ 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 [27], [28]. 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 [48] or TLSNotary [49]) 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 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 [29]).

### 7.3. System Design

As some features required to efficiently instantiate the KM-Buyer contract (Figure 6) are not yet available in Ethereum (see Section 7.2), 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 $F_{KM}$. 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 $S_i$ uses an enclave to attest to the correct encryption of her exploit under a symmetric key $k$, as well as to a commitment $\text{com}$ to $k$, both of which are sent off-chain to the buyer. An honest buyer sends $(S_i,\text{com})$ to a smart contract holding the reward, which $S_i$ 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 work 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 $\text{pro}(\text{inp}, r) = \text{true}$. Let $p := \Pr_r[\text{pro}(\text{inp}, r) = \text{true}]$ be the success probability of the exploit, for uniformly random $r$. Then, we can extend the $F_{SGP}$ functionality to compute an estimate $\tilde{p}$ of $p$, by running $\text{pro}(\cdot)$ for $n$ independent random strings. By a Hoeffding bound [50], the probability that $|p - \tilde{p}| > \delta$, for some constant $\delta$, is negligible in the security parameter $\lambda$ if $n = \Omega(\lambda)$. The buyer thus specifies a target probability $t$ and an allowed error $\delta$. 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.

### 7.4. 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 overhead compared to the cost of running the 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 [51]–[53]) to the SGX programming model required the addition of only a few hundred lines of “glue code”. In contrast, a system like Haven [21], capable of running arbitrary applications in SGX enclaves, is comprised of millions of line of code [21]. 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” [17]. 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 [39] 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
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 [56]. By crafting randomly “mutated” X.509 certificates, the authors revealed serious flaws in libraries’ certificate validation logic. Although SQL Injections are highly recognizable, and simple and efficient countermeasures exist, they remain one of the primary reported exploits in industry [55].

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.

<table>
<thead>
<tr>
<th>Application</th>
<th>&quot;Glue&quot; LOC</th>
<th>Time (ms)</th>
<th>Proof Size (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrapper</td>
<td>365</td>
<td>37</td>
<td>1152</td>
</tr>
<tr>
<td>SQL Injection</td>
<td>+166</td>
<td>2290(^{\dagger})</td>
<td>+166</td>
</tr>
<tr>
<td>X.509 Validation</td>
<td>+195</td>
<td>42</td>
<td>+220</td>
</tr>
</tbody>
</table>

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.

database. The HTTP processing makes use of Facebook’s Proxygen [54] and the database is built on SQLite [53]. Our application is vulnerable to a simple SQL Injection attack [55], 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 [55].

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.

8. Related Work

Isolated Execution Environments. The use of trusted hardware for secure computation dates back to platforms such as XOM [57], Aegis [16] or TPM-based solutions such as Flicker [18]. 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 [19], VC3 [20] and Haven [21]. 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]. Although side-channel attacks on memory integrity have been demonstrated across virtual machines [58], 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 [19]–[22], [29]. Others suggest more defensive architectures or programming paradigms, with the goal of eliminating or reducing data leakage from enclaves [3], [9], [11]–[15]. 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 [43], [44] or garbled circuits [59], [60] 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 tends to cover only a small subset of programs and bugs of interest.

**Bug Bounties.** Netscape launched the first bug bounty program in 1995 [61]. Today, bug bounties are widespread in industry, with a fast growth in rewarded discoveries and number of bounty hunters [62]. Illegal markets for 0-day exploits are also emerging, under the cover of anonymity provided by Tor or Bitcoin [63]. 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 [21].

**Cryptocurrencies and Fair-Exchange.** A number of works have used cryptocurrencies to incentivize fair behavior in multiparty computations, through threats of monetary penalties [64]–[66]. The use of Bitcoin transactions and smart contracts for the sale of information has been considered in [24], [25], [67]. 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**


Appendix A.
Proof of Theorem 1

Proof. Let $\mathcal{A}$ be a static adversary against ProtSGP in the $F_{TEE}$-hybrid model. We construct an ideal-world adversary (the simulator) Sim such that no environment $Z$ can distinguish between interactions with $\mathcal{A}$ and ProtSGP or with $\tilde{\mathcal{A}}$ and $F_{SGP}$.

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

Corrupt Prover: Consider the following sequence of hybrids, starting from a real-world execution of ProtSGP. Hybrid $H_1$ lets Sim emulate $F_{TEE}$. $H_2$ rules out forgery attacks on $\Sigma$. $H_3$ ensures that a single enclave is used throughout the protocol and $H_4$ ensures that the prover correctly forwards information between $F_{TEE}$ and $V$.

Finally, $H_5$ is equivalent to an ideal execution.

Hybrid $H_1$ proceeds as in the real protocol, except that Sim emulates $F_{TEE}$ by generating a keypair $(pk_{TEE}, sk_{TEE})$ for $\Sigma$ and publishing $pk_{TEE}$. When the adversary $\mathcal{A}$ wants to communicate with $F_{TEE}$, Sim records $\mathcal{A}$’s messages and faithfully emulates $F_{TEE}$’s behavior.

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

Hybrid $H_2$ is the same as $H_1$ except for the following modifications. If $\mathcal{A}$ invokes $F_{TEE}$ with a correct message $(\text{sid}, \text{idix}, \text{"install"}, \rho_{prop})$, then for every “resume” call to $\rho_{prop}$, Sim records a tuple $(\text{msg}, \sigma)$ where $\text{msg}$ is the output of $\rho_{prop}$, and $\sigma$ is the produced attestation under $sk_{TEE}$. Let $\Omega$ denote the set of all such tuples. Whenever $\mathcal{A}$ sends a signature $\sigma$ to $V$, let $\text{msg}$ be the message over which $V$ verifies $\sigma$ in ProtSGP. Then, if $(\text{msg}, \sigma) \notin \Omega$, Sim aborts.

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

Hybrid $H_3$ proceeds as in $H_2$, except for the following difference. When $\mathcal{A}$ sends $(\text{sid}, \text{"receipt"}, N, \sigma_c)$ to $V$, let $\text{idix}$ be the identifier of the enclave running $\rho_{prop}$ that produced $N$ and $\sigma_c$ (if no such $\text{idix}$ exists, Sim aborts). Then, if $\mathcal{A}$ ever sends a message to $V$ containing a signature $\sigma$ generated by an enclave with a different identifier $\text{idix}' \neq \text{idix}$, Sim aborts.

First note that if an $\text{idix}$ as defined above doesn’t exist, then $V$’s signature verification will fail (i.e., the nonce sent by $\mathcal{A}$ does not match the one signed by $\rho_{prop}$). As $F_{TEE}$ samples new randomness on every “resume” call, launching multiple enclaves running $\rho_{prop}$ will produce a new nonce $N' \neq N$ with overwhelming probability, implying that $V$’s signature verification for $\sigma_r$ or $\sigma_o$ will fail. $H_3$ and $H_2$ are thus indistinguishable.

Hybrid $H_4$ proceeds as in $H_3$ with the following differences. When $\mathcal{A}$ sends $(\text{sid}, \text{"receipt"}, N, \sigma_c)$ to $V$, Sim records the input $\text{inp}_P$ that $\mathcal{A}$ sent to $F_{TEE}$ to obtain $\sigma_c$. If the verifier submits a challenge $\text{inpp}_V$, Sim forwards it to $\mathcal{A}$. When $\mathcal{A}$ sends $(\text{sid}, \text{"proof"}, \text{outp}, \sigma_r)$ to $V$, Sim aborts if $\text{outp}$ and $\text{sr}$ were not obtained from a “resume” call to $\rho_{prop}$ with input (“prove”, $\text{inpp}_V$) (where $\text{inpp}_V = \perp$ if $V$ sent no challenge). Sim also aborts if $\mathcal{A}$ sends a message $(\text{sid}, \text{"open"}, \text{inp}, \sigma_o)$ to $V$ where $\text{inp} \neq \text{inpp}_P$.

$H_4$ and $H_3$ are trivially indistinguishable, as in both of the described abort cases, $V$’s signature verification fails.

Hybrid $H_5$ is as $H_4$, except for the following differences: When $\mathcal{A}$ sends $(\text{sid}, \text{"receipt"}, N, \sigma_c)$ to $V$, Sim finds the recorded input $\text{inp}_P$ that $\mathcal{A}$ sent to $F_{TEE}$ to obtain $(N, \sigma_c)$, and sends $(\text{sid}, \text{"commit"}, \text{inpp}_P)$ from $P$ to $F_{SGP}$ in the ideal-world. When $\mathcal{A}$ asks $F_{TEE}$ to resume $\rho_{prop}$ with input (“prove”, $\text{inpp}_V$), Sim sends $(\text{sid}, \text{"prove"})$ to $F_{SGP}$ on $P$’s behalf, and obtains $(\text{sid}, \text{"proof"}, \text{outp}, r)$ from $F_{SGP}$. To emulate $F_{TEE}$’s response to $\mathcal{A}$, Sim then simply uses this randomness $r$. When $\mathcal{A}$ sends $(\text{sid}, \text{"proof"}, \text{outp}, \sigma_r)$ to $V$, Sim lets $F_{SGP}$ deliver the output to $V$ in the ideal world. Finally, when $\mathcal{A}$ sends $(\text{sid}, \text{"open"}, \text{inp}, \sigma_o)$ to $V$, Sim sends $(\text{sid}, \text{"open"})$ to $F_{SGP}$.

It is immediate that $\mathcal{A}$’s view in $H_4$ is distributed identically from the view in $H_5$, as the only difference is that in $H_4$ the randomness $r$ is randomly sampled by Sim, and in $H_5$ it is sampled by $F_{SGP}$.

It remains to observe that in $H_5$, Sim can interact directly with $F_{SGP}$ in the ideal-world, faithfully emulate $\mathcal{A}$’s view in the real-world, and then output exactly what $\mathcal{A}$ outputs. Thus, no environment $Z$ can distinguish an interaction with $\mathcal{A}$ and the real protocol ProtSGP from an interaction with Sim and ProtSGP.

Honest Prover: Simulating the view of a network adversary or of a corrupt verifier is considerably simpler, as $V$ sends at most a single message in ProtSGP. We directly describe Sim’s behavior:

- Upon receiving a message $(\text{sid}, \text{"commit"})$ from $F_{SGP}$, Sim sends a valid message $(\text{sid}, \text{"receipt"}, N, \sigma_c)$ to $V$ (whether honest or corrupt), for a randomly chosen nonce $N$, and a correct attestation $\sigma_c$.
- If $V$ is honest and Sim gets $(\text{sid}, \text{"challenge"}, \text{inpp}_P)$ from $F_{SGP}$, Sim simply emulates a message $(\text{sid}, \text{"challenge"}, \text{inpp}_P)$ to $P$. Otherwise, if $V$ is corrupted and $\mathcal{A}$ sends $(\text{sid}, \text{"challenge"}, \text{inpp}_P)$ to the honest $P$ on $V$’s behalf, Sim sends $(\text{sid}, \text{"challenge"}, \text{inpp}_P)$ to $F_{SGP}$ on $V$’s behalf.
- If Sim receives $(\text{sid}, \text{"proof"}, \text{outp})$ from $F_{SGP}$, Sim emulates a message $(\text{sid}, \text{"proof"}, \text{outp}, \sigma_r)$ from $P$ to $V$, where $\sigma_r$ is a valid signature under $sk_{TEE}$ over $\text{outp}_V$, $\text{outp}$, and $N$.
- Finally, when the commitment to $F_{SGP}$ is opened, Sim obtains $(\text{sid}, \text{"open"}, \text{inpp}_P)$ from $F_{SGP}$ and can emulate a valid opening $(\text{sid}, \text{"open"}, \text{inpp}_P, \sigma_o)$ to $V$, by creating a valid signature $\sigma_o$ over $\text{inpp}_P$ and $N$.

The simulation of $\mathcal{A}$’s view is perfect: $V$ obtains valid digital signatures $\sigma_c, \sigma_r, \sigma_o$ from $P$, that are linked by a
randomly chosen nonce \( N \). The real-worlds and ideal-world executions are thus indistinguishable to \( Z \).

\[ \square \]

**Appendix B.**

**Proof of Theorem 3**

Proof: Let \( A \) be a static adversary against Prot\(_{\text{KM}}\) in the \( \mathcal{F}_{\text{SGP}} \)-hybrid model. We construct an ideal-world simulator \( \text{Sim} \), such that no environment \( Z \) can distinguish between interactions with \( A \) and Prot\(_{\text{KM}}\) or with \( \text{Sim} \) and \( \mathcal{F}_{\text{KM}} \). We first describe the simulator \( \text{Sim} \), which internally runs a copy of the real-world adversary \( A \). Note that the wrappers \( \mathcal{F}, \mathcal{G} \) and \( \Pi \) around ideal contracts, real-world contracts and protocols, defined in [42], are implicit.

In each round, the simulation generically proceeds as follows: If a message is sent by an honest party to \( \mathcal{F}_{\text{KM}} \), \( \text{Sim} \) obtains information from the “immediate actions” executed by \( \mathcal{F}_{\text{KM}} \). \( \text{Sim} \) then appropriately emulates the same actions for its internal copy of \( A \), as described in more detail for each message type below. If messages are sent to KM-Buyer on behalf of corrupt parties, \( \text{Sim} \) records those messages and awaits the end of the round. Then, \( \text{Sim} \) obtains the order chosen by \( A \) for processing messages in KM-Buyer. For those messages that pass KM-Buyer’s “assert” statements, \( \text{Sim} \) sends an appropriate message on behalf of a corrupted party to \( \mathcal{F}_{\text{KM}} \) (details below), and instructs \( \mathcal{F}_{\text{KM}} \) to process these messages in an order that is consistent with the real-world execution.

We now provide further details on the processing of specific messages in each round:

1. **Initialization Phase:**
   - \( \text{Sim} \) obtains (\( \text{sid}, \text{"init"}, B, \text{\$Reward}, \text{timeout} \)) by querying \( \mathcal{F}_{\text{KM}} \)'s state if the Buyer \( B \) is honest. It then faithfully emulates \( B \)'s role in Prot\(_{\text{KM}}\) by generating a keypair \((pk_B, sk_B)\) and emulating an “init” message to KM-Buyer.
   - If the Buyer \( B \) is corrupt, \( \text{Sim} \) sends (\( \text{sid}, \text{"init"}, prog, \text{\$Reward}, \text{timeout}, pk_B \)) when \( A \) sends it to KM-Buyer. On behalf of \( B \), \( \text{Sim} \) then sends (\( \text{sid}, \text{"init"}, prog, \text{\$Reward}, \text{timeout} \)) to \( \mathcal{F}_{\text{KM}} \).
   - In both cases, \( \text{Sim} \) will emulate interactions between KM-Buyer and \( \mathcal{F}_{\text{SGP}} \), on behalf of \( A \) or honest parties.

2. **Commitment Phase:**
   - If an honest Seller \( S_i \) is invoked with input (\( \text{sid}, \text{"commit"}, \text{inp}_{S_i} \)) \( \text{Sim} \) obtains (\( \text{"receipt"}, S_i, T \)) by querying \( \mathcal{F}_{\text{KM}} \)'s state. Then, \( \text{Sim} \) emulates a message (\( S_i, \text{"commit"} \)) to KM-Buyer on behalf of \( \mathcal{F}_{\text{SGP}} \) \((\text{prog}, S_i, \text{KM-Buyer})\). At the end of the round, \( \text{Sim} \) emulates KM-Buyer’s processing of the message.
   - If a corrupt Seller \( S_j \) sends a message to \( \mathcal{F}_{\text{SGP}} \), \( \text{Sim} \) records the message and faithfully emulates \( \mathcal{F}_{\text{SGP}} \)'s response. If \( S_j \) sends (\( S_j, \text{"commit"}, (\text{inp}_{S_j}, pk_B) \)) to \( \mathcal{F}_{\text{SGP}} \) \((\text{prog}, S_j, \text{KM-Buyer})\), where \( pk_B \) is the Buyer’s public key, \( \text{Sim} \) sends (\( \text{sid}, \text{"commit"}, \text{inp}_{S_i} \)) to \( \mathcal{F}_{\text{KM}} \) on behalf of the corrupt \( S_j \).

3. **Claiming Phase:**
   - The ordering of “claim” messages in a round may determine which successful seller receives the reward. At the end of a round, \( \text{Sim} \) thus chooses the same ordering of messages fo \( \mathcal{F}_{\text{KM}} \) as the one chosen by \( A \) for KM-Buyer (ignoring messages that are rejected by KM-Buyer).
   - If an honest seller \( S_i \) sends (\( \text{sid}, \text{"claim"} \)) to \( \mathcal{F}_{\text{KM}} \) but the evaluation of \( \text{prog}(\text{inp}_{S_i}, r) \) fails, \( \text{Sim} \) learns nothing from \( \mathcal{F}_{\text{KM}} \). This models the real-world where the adversary learns nothing if an honest seller interacts with \( \mathcal{F}_{\text{SGP}} \) (and thus \( \mathcal{F}_{\text{TEE}} \)) and the evaluation of prog fails.
   - If an honest seller \( S_i \) sends (\( \text{sid}, \text{"claim"} \)) to \( \mathcal{F}_{\text{KM}} \) and the evaluation of prog succeeds, \( \text{Sim} \) obtains (“claimed”, \( S_i \)) by querying \( \mathcal{F}_{\text{KM}} \)'s state if \( B \) is honest, and also \( \text{inp}_{S_i} \) if \( B \) is corrupt. \( \text{Sim} \) computes \( c := \text{Enc}_{pk_B}(\text{inp}_{S_i}) \) if \( B \) is corrupt, and \( c := \text{Enc}_{pk_B}(0) \) otherwise. In both cases, \( \text{Sim} \) generates a message (\( \text{sid}, \text{"proof"}, (c, pk_B) \)) on behalf of \( \mathcal{F}_{\text{SGP}} \) \((\text{prog}, S_i, \text{KM-Buyer})\) to KM-Buyer.
   - If a corrupt Seller \( S_j \) sends (\( S_j, \text{"prove"} \)) to \( \mathcal{F}_{\text{SGP}} \) \((\text{prog}, S_j, \text{KM-Buyer})\), \( \text{Sim} \) emulates \( \mathcal{F}_{\text{SGP}} \) as follows: If \( S_j \) had previously correctly committed to an input \( \text{inp}_{S_j} \), \( \text{Sim} \) sends (\( \text{sid}, \text{"claim"} \)) to \( \mathcal{F}_{\text{KM}} \) on \( S_j \)'s behalf (here \( \text{Sim} \) does not wait until the end of the real-world round to interact with \( \mathcal{F}_{\text{KM}} \), as it needs to obtain the randomness \( r \) from \( \mathcal{F}_{\text{KM}} \) to proceed with the simulation). If the ideal contract is in a correct state, \( \text{Sim} \) obtains \( r \). \( \text{Sim} \) then faithfully emulates \( \mathcal{F}_{\text{SGP}} \)'s response to the “prove” call, with randomness \( r \). If the evaluation of \( \text{prog} \) succeeds, \( \text{Sim} \) sends (\( S_j, \text{"proof"}, (c, pk_B) \)) to KM-Buyer on \( \mathcal{F}_{\text{SGP}} \)'s behalf, where \( c := \text{Enc}_{pk_B}(\text{inp}_{S_j}) \) if \( B \) is corrupt, and \( c := \text{Enc}_{pk_B}(0) \) otherwise. \( \text{Sim} \) then applies the same processing order to \( \mathcal{F}_{\text{KM}} \) as \( A \) chooses for KM-Buyer. If at least one seller (honest or corrupt) correctly claimed the reward in this round, the first seller \( S_i \) to have her claim processed will obtain the reward.

4. **Timer:** Note that the timer functionality is the same in the real and ideal worlds and can thus be simulated perfectly. Specifically, whenever the environment instructs an honest party to send a tick to \( \mathcal{F}_{\text{KM}} \) in the ideal world, \( \text{Sim} \) is notified of this and can emulate sending a tick to KM-Buyer on behalf of the honest party in the real world [42].

5. **Internal State:** On any call (\( \text{sid}, \text{"get-state"} \)) to KM-Buyer, \( \text{Sim} \) simply responds by first querying \( \mathcal{F}_{\text{KM}} \)'s state, and crafting appropriate “extra data” \( D' \). Specifically, after KM-Buyer has been initialized, \( D' \) contains (\( \text{prog}, pk_B \)) which \( \text{Sim} \) obtained from \( A \). After a successful claim, \( D' \) further contains the simulated ciphertext \( e \) produced by \( \text{Sim} \).

Consider a sequence of hybrids, starting from a real-world execution of Prot\(_{\text{KM}}\). Hybrid \( H_1 \) lets \( \text{Sim} \) emulate \( \mathcal{F}_{\text{SGP}} \) and KM-Buyer. \( H_2 \) has \( \text{Sim} \) emulate the initialization and commitment phases. \( H_3 \) replaces the random coins of \( \mathcal{F}_{\text{SGP}} \) by those of \( \mathcal{F}_{\text{KM}} \), and \( H_4 \) replaces ciphertexts under \( pk_B \) by encryptions of 0, when \( B \) is honest.
Hybrid $H_1$ proceeds as in the real protocol, except that Sim emulates $\mathcal{F}_{SGP}$ and KM-Buyer. When the adversary $A$ wants to communicate with $\mathcal{F}_{SGP}$ or KM-Buyer, Sim records $A$'s messages and faithfully emulates $\mathcal{F}_{SGP}$'s or KM-Buyer’s behavior.

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

Hybrid $H_2$ is the same as $H_1$, except that Sim emulates the initialization and commitment phases. As described above, if $B$ is honest, Sim generates $(pk_B, sk_B)$ and initializes KM-Buyer. If $B$ is corrupt, Sim simply initializes $\mathcal{F}_{KM}$ on $B$’s behalf, using the information provided by $A$. In the commitment phase, Sim extracts inputs committed by corrupt parties when sent to $\mathcal{F}_{SGP}$, and forwards them to $\mathcal{F}_{KM}$. When an honest seller commits in the ideal-world, Sim simply emulates a valid acknowledgement on behalf of $\mathcal{F}_{SGP}$, and updates the state of the emulated KM-Buyer contract.

It is trivial to see that $A$’s view in $H_2$ is distributed exactly as in $H_1$, by the definition of $\mathcal{F}_{SGP}$.

Hybrid $H_3$ is the same as $H_2$ except for the following modification. When a corrupt Seller $S_j$ sends $(S_j, \text{"prove"})$ to $\mathcal{F}_{SGP}[\mathsf{prog}, S_j, \text{KM-Buyer}]$, Sim checks that $S_j$ had previously committed a value $\mathsf{inp}_{S_j}$, and otherwise ignores the message. Then, Sim sends $(\text{sid, \"claim"})$ to $\mathcal{F}_{KM}$ on $S_j$’s behalf. If the ideal contract is in a correct state, Sim obtains $r$, and emulates $\mathcal{F}_{SGP}$’s computation of $\mathsf{prog}$, with randomness $r$. Otherwise, Sim emulates $\mathcal{F}_{SGP}$ by sampling $r$ on its own, and sending the response to KM-Buyer.

We observe that $A$’s view in $H_3$ is distributed exactly as in $H_2$: if $\mathcal{F}_{KM}$ rejects $S_j$’s message, KM-Buyer would also reject a message from $\mathcal{F}_{SGP}[\mathsf{prog}, S_j, \text{KM-Buyer}]$, as the timer and state of $\mathcal{F}_{KM}$ and KM-Buyer are identical throughout the simulation. Finally, if $\mathcal{F}_{KM}$ accepts the message, the only difference between $H_2$ and $H_3$ is that in $H_2$ the randomness used in $\mathcal{F}_{SGP}$ is sampled by Sim, and in $H_3$, it is sampled by $\mathcal{F}_{KM}$.

Hybrid $H_4$ proceeds as in $H_3$, except for the following difference. If the Buyer is honest, then Sim replaces any message $(\text{sid, \"proof"}, (c, pk_B))$ sent by $\mathcal{F}_{SGP}$ to KM-Buyer, by a message $(\text{sid, \"proof"}, (c', pk_B))$, where $c' := \mathsf{Enc}_{pk_B}(0)$. In later rounds, $\mathcal{F}_{SGP}$ uses the same randomness as $\mathcal{F}_{KM}$ to run $\mathsf{prog}$, and thus produces a valid proof for KM-Buyer exactly when a claim succeeds in $\mathcal{F}_{KM}$ as well.

If the Buyer is corrupt, the adversary in $H_4$ obtains a valid ciphertext for the Seller (whether honest or corrupt) that submits a successful claim, and can thus extract a valid input. If the adversary had previously committed to a valid input on behalf of a corrupted party, it can still claim the actual reward, as also allowed in $\mathcal{F}_{KM}$, by selecting the message processing order.

Finally, timeouts in $\mathcal{F}_{KM}$ and KM-Buyer occur exactly in the same round, and result in the Buyer recovering her reward if no successful claim was made.