Succinct Scriptable NIZK via Trusted Hardware

Bingsheng Zhang[1], Yuan Chen[1], Jiaqi Li[1], Yajin Zhou[1], Phuc Thai[2], Hong-Sheng Zhou[2], and Kui Ren[1,3]

[1] Zhejiang University, {bingsheng,yajin_zhou}@zju.edu.cn
[2] Virginia Commonwealth University, {thaipd,hszhou}@vcu.edu
[3] Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, kuiren@zju.edu.cn

Abstract. Non-interactive zero-knowledge proof or argument (NIZK) systems are widely used in many security sensitive applications to enhance computation integrity, privacy and scalability. In such systems, a prover wants to convince one or more verifiers that the result of a public function is correctly computed without revealing the (potential) private input, such as the witness. In this work, we introduce a new notion, called succinct scriptable NIZK, where the prover and verifier(s) can specify the function (or language instance) to be proven via a script. We formalize this notion is UC framework and provide a generic trusted hardware based solution. We then instantiate our solution in both SGX and Trustzone with Lua script engine. The system can be easily used by typical programmers without any cryptographic background. The benchmark result shows that our solution is better than all the known NIZK proof systems w.r.t. prover's running time (1000 times faster), verifier's running time, and the proof size. Finally, we show how the proposed scriptable succinct NIZK can be readily deployed to solve many well-known problems in the blockchain context, e.g. verifier's dilemma, fast joining for new players, etc.

1 Introduction

Collaboration is one of the main driving forces for the sustainable advancement of our civilization, growing from small-size tributes, to cities, and then to large-scale states. Being a part of the modern society, we are interacting with hundreds of known/unknown entities either physically or remotely. The main motivation of this work is to introduce new concepts and frameworks to enable more effective collaborations. One potential candidate tool is a well-known cryptographic primitive—zero knowledge (ZK) proof/argument system. In a ZK system, two players, the prover and the verifier, are involved; one one hand, the prover who holds a valid witness of an NP statement, is able to convince the verifier that the statement is true without revealing the corresponding witness; on the other hand, if the prover does not know any valid witness of the statement, then he cannot convince the verifier. ZK systems can be used to enable trustworthy collaborations: all players in a protocol are required to prove the correctness of their behaviors in the protocol execution. However, to enable effective collaborations, desired properties are expected, and we will elaborate them below.

Our design choices. In a large-scale collaboration network, it is infeasible for a party to prove the correctness of its computation to all other parties one by one. The first property we need from ZK systems, is *(1) non-interactiveness* in the sense that the prover only needs to prove the correctness of the computation once, and the prover then can send the same proof to all other parties i.e., the verifiers. From now on, we use NIZK to denote non-interactive ZK systems. The second desirable property we need is *(2) succinctness*, given the fact that the bottleneck for large-scale collaboration is the capacity of the underlying peer-to-peer network communication. Furthermore, as already mentioned, we note that in a typical application scenario a single prover will prove the same statement to many verifiers. In this *unbalanced* setting, a desirable NIZK proof system should have the property of *(3) lightning fast verifier*.

Up to now, those properties have already been achieved by a number of existing NIZK proof systems, such as zk-SNARK [16, 34], zk-STARK [3], etc. However, these NIZK systems have not been widely used in practice yet. A significant barrier is the that the computation of prover is very heavy. The state-of-the-art NIZK systems need hours to prove large statement even on a powerful PC (32 cores and 512 GB RAM [3]), let alone portable devices such as smartphones, tablets, and IoT devices. We aim to develop a NIZK system with the property of *(4) truly lightweight prover*.

To enable wide adoption of NIZK in the real world, the design must be *(5) deployment friendly*. The underlying cryptographic machinery should be transparent to the developers, and the protocol can be operated without cryptographic background. Unfortunately, all existing NIZK proof systems for universal language require re-compilation of both prover and verifier's executable binary files for every new language instance.

Our approach. We propose a new primitive, called *succinct scriptable NIZK*, with the goal of achieving all desirable properties above. This new primitive allows the developers to specify the language instance or computation to be verified *via a script without any re-compilation*. Similar to NIZK proof systems for universal language, a scriptable NIZK system can support multiple language instances, depending on the script language design and the script engine execution environment. Different from existing succinct NIZK systems for universal language, our scriptable NIZK is very easy to use; for a new language instance, the players can easily define the scripts and no further compilation is required.

Defining scriptable NIZK. We assume both the prover and the verifiers have agreed on the *function/script*, denoted as $\mathcal{C}$, the *public input*, denoted as $\mathsf{Input}_{\mathrm{pub}}$, and the *(public) output*, denoted as Output; in addition, the prover keeps a *private input*, denoted as $\mathsf{Input}_{\mathrm{priv}}$, such that $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output}$. The prover is able to prove to the verifiers that he knows a private input $\mathsf{Input}_{\mathrm{priv}}$ that would make the script execution $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ to generate output Output.

We note that not all scripts can be supported; each scriptable NIZK system is parameterized by a predicate Q, and $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}) = 1$ for any valid script $\mathcal{C}$. The predicate Q is defined by the script language design and the script engine execution environment.

An NP language $\mathcal{L}$ is defined by its polynomial-time decidable relation $\mathcal{R}$; namely, $\mathcal{L} := \{x : \exists w \text{ s.t. } (x, w) \in \mathcal{R}\}$. In practice, for each relation $\mathcal{R}$, we assume there exists a corresponding script $\mathcal{C}_{\mathcal{R}}$ such that $\mathcal{C}_{\mathcal{R}}(x, w) = 1$ iff $(x, w) \in \mathcal{R}$; otherwise, $\mathcal{C}_{\mathcal{R}}(x, w) = 0$. To use the scriptable NIZK system for an NP language, the prover and the verifiers set $\mathsf{Input}_{\mathrm{pub}} := x$, $\mathsf{Input}_{\mathrm{priv}} := w$, $\mathsf{Output} := 1$, and the script as $\mathcal{C}_{\mathcal{R}}$. The notion is formally modeled in the UC framework.

Constructing succinct scriptable NIZK. We then present a generic succinct scriptable NIZK construction in the trusted hardware model. Trusted hardware can enable an isolated and trusted computation environment where security sensitive data can be stored and processed with confidentiality and integrity guarantees. Most existing trusted hardware based applications, e.g., [15] emphasize on the confidentiality aspect, while the security of our construction mainly relies on the computational integrity guaranteed by trusted hardware. The main idea is as follows. Recall that in a NIZK proof, the prover and the verifier have common input $(\mathcal{C}_{\mathcal{R}}, \mathsf{Input}_{\mathrm{pub}} := x)$. The potentially malicious prover wants to convince the verifiers that he knows a witness $\mathsf{Input}_{\mathrm{priv}} := w$ such that $\mathcal{C}_{\mathcal{R}}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = 1$. Since the trusted hardware can guarantee computation integrity even when the host is malicious, we can let $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ to execute the relationship decision algorithm $b \leftarrow \mathcal{C}_{\mathcal{R}}(x, w)$ and sign the output b. To bind the decision algorithm and statement, we let $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ sign $(\mathcal{C}_{\mathcal{R}}, x, b)$ without revealing the witness w. Therefore, by checking the signature, the verifier is convinced that the prover must know a witness w such that $\mathcal{C}_{\mathcal{R}}(x, w) = 1$ if $(\mathcal{C}_{\mathcal{R}}, x, 1)$ is signed by $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$. Similarly, for general computation, the private input $\mathsf{Input}_{\mathrm{priv}}$ is not signed; therefore, zero-knowledge property is preserved even if the signature leaks the signed message.

Although there are a number of works in the literature studying how to speed up secure computing via trusted hardware, such as Intel SGX, we emphasize that this problem has not been solved by previous works. The closest related work is sealed-glass proof introduced by Tramer *et al.* [40], where the authors try to explore some use cases even if the isolated execution environment has unbounded leakage, i.e., arbitrary side-channels. We note that, their primitive is interactive, thus not scalable; in their protocol, for each verification, the trusted hardware must be interacted with. Our primitive is non-interactive, and in our construction, the verifier can verify the proof without interacting with the trusted hardware. There are also many theoretical differences between interactive ZK and non-interactive ZK, such as the minimum assumptions needed to realize the primitive; therefore, this work is not covered by [40]. Most importantly, ours is the first work to investigate *scriptable* NIZK, which is developer-friendly.

Instantiation. We instantiate our succinct scriptable NIZK proof system on two most popular trusted hardware platforms: Intel SGX and Arm TrustZone. The main component is the Q-compliant hardware functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$. In terms of Intel SGX, the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ functionality is instantiated by three entities: the (trusted) Intel server, the prover, and the SGX hardware device. In terms of Arm TrustZone, currently only manufacture has the privilege to access TrustZone root keys; nevertheless, our system uses Hikey 960 TrustZone development board. The $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ functionality is instantiated by two entities: the (trusted) authority server, and the TrustZone development board.

With regard to scriptability, in practice, it is a challenge for a third party to verify the consistency between an executable binary and its software specification. That is, the binary contains no bug, no trapdoor, and it is not subverted. Even if it is possible, it dramatically increases the verifier's complexity. On the other hand, it is implausible to assume a trusted third party that is available to generate a certified binary for each language instance. To address this issue, we decide to adopt a scripting language, called Lua. Lua is a lightweight script language. We implemented modified Lua script engine for both Intel SGX enclave computation environment and the TrustZone environment. At a high level, we let the Intel server and/or the setup authority server to prepare and sign a Lua engine enclave/binary. The signed Lua engine is published as a common reference string (CRS). In addition, the hardware is initialized with a signing key, and it corresponding public key is also published as a part of the CRS. The modified Lua engine takes input as a script $\mathcal{C}$, a public input $\mathsf{Input}_{\mathrm{pub}}$, a private input $\mathsf{Input}_{\mathrm{priv}}$, and a tag tag that can be used to store auxiliary information, such as session id. The Lua engine runs $\mathsf{Output} \leftarrow \mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ and signs $\langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output} \rangle$. Therefore, any verifier who has the public key can verify the signature. The predicate Q is restricted by the Lua engine constrain. For instance, there is a fixed heap size, e.g., 32MB when the Lua engine enclave is built. It limits the maximum script size. Moreover, as security requirement, one may want to introduce a maximum running time to prevent the script from running forever. Such a running time cap would also reflected by Q.

Recall that scriptable NIZK proofs are typically deployed in a one-to-many scenario, where the prover only needs to invoke the trusted hardware once and many verifiers can check the validity of the proof; however, currently, the remote attestation of Intel's SGX requires the verifier to interact with the Intel Attestation Service (IAS) server. If each verifier needs to query the Intel IAS server to check the proof, the overall performance is limited by the throughput of Intel's IAS. Moreover, the validity of a NIZK proof should be consistent over time, i.e., if a NIZK proof is verifiable at this moment, the same proof should remain verifiable in the future. Unfortunately, this would not be the case if we invoke the Intel IAS in the verification process; certifying an old quote (say, generated 1 year ago) is never the design goal of Intel's remote attestation. This is because the quote needs to contain a non-revoked proof for each item on the signature revocation list, and the proof is no longer verifiable once the revocation list is updated at the Intel side. That means a quote is only valid until the next revocation list update. To resolve this issue, in our design, after generating the quote, the prover immediately queries the Intel IAS server for the attestation verification report on behave of a verifier. Since the attestation verification report is signed by Intel, given Intel's public key, anyone can verify the validity of the attached signature. This tweak also makes the verification process non-interactive.

Performance. The performance of our succinct scriptable NIZK system is theoretically and experimentally evaluated and compared with the other NIZK proof systems. Table 1 illustrates the asymptotic efficiency comparison measured by the circuit size. $|C|$ is the circuit size; $|w|$ is the witness size; $|c|$ is the problem instance size; s is the number of copies of the subcircuits; d is the width of the subcircuits. As we can see, our construction can achieve constant CRS size, constant verifier's complexity, and constant proof size. The prover's complexity is also minimum, which is $|C|$. Note that in theory, the verifier's complexity cannot be sublinear to the statement size $|x|$, but as a convention, it is ignored in the table.

Scheme	Setup size	Proof size	Prover's time	Verifier's time	Setup Asm.	Comp. Asm.
Ligero	1	$\sqrt{\lvert C\rvert}$	$\lvert C\rvert \log \lvert C\rvert$	$s \log s + d \log d$	RO	CRHF
Bootle *et al.*	1	$\sqrt{\lvert C\rvert}$	$\lvert C\rvert$	$\lvert C\rvert$	RO	CRHF
Baum *et al.*	$\sqrt{\lvert C\rvert}$	$\sqrt{\lvert C\rvert \log \lvert C\rvert}$	$\lvert C\rvert \log \lvert C\rvert$	$\lvert C\rvert$	CRS	SIS
zk-STARKs	1	$\log^2 \lvert C\rvert$	$\lvert C\rvert\, \mathsf{polylog}(\lvert C\rvert)$	$\mathsf{polylog}(\lvert C\rvert)$	RO	CRHF
Aurora	1	$\log^2 \lvert C\rvert$	$\lvert C\rvert \log \lvert C\rvert$	$\lvert C\rvert$	RO	CRHF
Bulletproof	$\lvert C\rvert$	$\log \lvert C\rvert$	$\lvert C\rvert \log \lvert C\rvert$	$\lvert C\rvert \log \lvert C\rvert$	CRS + RO	DL
SNARKs	$\lvert C\rvert$	1	$\lvert C\rvert \log \lvert C\rvert$	$\lvert c\rvert$	CRS/AGM	KE
This work	1	1	$\lvert C\rvert$	1	HW	Signature

Table 1: Asymptotic efficiency comparison of different NIZK proof/argument systems. $|C|$ is the circuit size; $|w|$ is the witness size; $|c|$ is the problem instance size; s is the number of copies of the subcircuits; d is the width of the subcircuits. DL stands for discrete logarithm assumption, CRHF stands for collision-resistant hash functions, SIS stands for shortest integer solution assumption, KE stands for knowledge-of-exponent assumption, HW stands for trusted hardware model, and AGM stands for algebraic group model.

In terms of the actual experimental performance. The prover's running time for evaluating a Boolean circuit consisting of 2^{39} NAND gates only takes less than 10 mins, which is 900 times faster than the state of the art, zk-STARK, for circuits larger than 2^{35} gates. Note that this performance result is tested through Lua script, and native code for circuit evaluation is 10 times faster in our experiment. The verifier's running time is merely a signature verification, which takes approximately 1.5 ms – better than all the other existing succinct NIZK systems. The proof size is 297 Bytes with current Intel SGX signature, where 256 Bytes are the signature. Hence, we envision it is possible to further reduce the proof size by replacing the signature scheme. The TrustZone based system uses ECDSA on the `secp256k1` curve, so the proof size is only 32 Bytes.

Applications. Finally, we discuss applications of our succinct scriptable NIZK. We note that, many applications have been previously investigated. However, it is very challenging to deploy them in practice due to the performance barrier.

Sound and scalable blockchain. As discussed at the very beginning of the Introduction, lots of heated discussions are taking place in blockchain community, with the goal of improving the performance in a sound manner. This consists of two parts. First, we should address the existing issues, since many blockchain scalability proposals have been implemented even the community is aware of the security concerns. Again, we note that, these issues were not addressed probably due to the missing of fast and succinct NIZK.

Second, we will enable new design paradigm for the interesting "one-to-many" unbalanced computation scenarios. Using our NIZK, typically, a single node as prover, can generate in very short time a proof that will convince all other nodes to accept the validity of the current state of the ledger, without requiring those nodes to naively re-execute the computation, nor to store the entire blockchain's state, which would be required for such a naive verification.

Privacy preserving smart contracts. The zero-knowledge properties of ZK proofs has already been intensively used in blockchain projects, with the goal of ensuring the anonymity and protecting financial privacy. Notably, Succinct Non-interactive ARguments of Knowledge (zk-SNARK) has been used in Zcash and Ethereum; Bulletproofs has been used in Monaro. Recently, Ethereum has the plan to explore the feasibility zk-STARK in its future version of their platform. We note that, it is still not clear if zk-STARK can be widely adopted in blockchain platforms given the fact that, the current proof size is $1000\times$ longer than zk-SNARKs. Fortunately, our NIZK is super succinct, and super fast.

2 Preliminaries

Trusted execution environment. Trusted execution environment (TEE) refers to a range of technologies that can establish an isolated and trusted environment where security sensitive data can be stored and processed with confidentiality and integrity guarantees. TEE needs to be instantiated on top of a trusted computing base (TCB), which consists of hardware, firmware and/or software. Minimizing the size (attack surface) of TCB with reasonable assumptions is the common goal of this line of research. In practice, TEE can be realized on top of several promising trusted hardware technologies, such as ARM TrustZone and Intel SGX. Although recently a few side-channel attacks, e.g. [29, 10], have been explored against those TEE candidates, new designs and fixes are proposed on a monthly basis. Hence, we envision that TEE will be a cheap and acceptable assumption in the near future. In this work, our benchmarks are mainly based on the Intel SGX platform for its readily deployed remote attestation infrastructure; however, our technique can also be implemented on any other TEE solutions.

Intel SGX. Intel Software Guard Extensions (SGX) is a widely used trusted hardware solution to enable TEE. It provides a hardware enforced isolated execution environment against malicious OS kernels and supervisor software. The SGX processor sets aside an exclusive physical memory space, called processor reserved memory (PRM) to ensure the confidentiality and integrity of enclave's memory. Each SGX hardware holds two root keys: root provisioning key and root seal key. The actual attestation keys are deviated from those root keys via PRF. Intel's (anonymous) attestation is based on an anonymous group signature scheme called Intel Enhanced Privacy ID (EPID) [9]. In this work, we are particularly interested in SGX's ability to enable attested computation, i.e. any third party can audit an outcome is computed by a pre-agreed program in a genuine SGX. More specifically, the application enclave first uses $\mathsf{EREPORT}$ to generate a report for *local attestation* (identifying two enclaves are running on the same platform). The report is then sent to a special enclave called Quoting Enclave (QE) to produce a *quote* by signing the report with the group signature. In theory, given the group public key (and the up-to-date revocation list), any verifier can check the validity of the signed quote non-interactively; however, currently, one must contact the Intel Attestation Service (IAS) for verification. IAS will first verify the group signature and then create the corresponding attestation verification report with its own signature.

NIZK proof/argument systems. Let $\mathcal{R}$ be a polynomial time decidable binary relation. We call x the statement and w the witness, if $(x, w) \in \mathcal{R}$. $\mathcal{L} := \{x \mid \exists w : (x, w) \in \mathcal{R}\}$ is the NP language defined by $\mathcal{R}$. In a zero-knowledge (ZK) proof/argument system, the prover wants to convince one or more verifier(s) $x \in \mathcal{L}$, where $\mathcal{L}$ is an arbitrary NP language. The ZK system is called non-interactive (NIZK) [7] if the prover can generate the proof without interacting with a verifier, and any verifier(s) can check the validity of the proof. However, it is not possible to realize a NIZK proof/argument system unless the language is in BPP in the plain model (a.k.a. standard model) [19]. To circumvent this impossibility result, all NIZK proof/argument systems must rely on some trusted setup assumptions, such as the common reference string model, random oracle model, and generic group model, etc. A NIZK system is called *succinct* if the proof size is asymptotically less than $|w| + |x|$ (cf. Sec. 3). Unfortunately, it is also shown in [17] that succinct NIZK proof/argument systems cannot be based on any falsifiable assumptions, i.e. an assumption that can be written as a game. That means one must embrace "strong assumptions" to enjoy the benefit of succinctness. In addition, many NIZK proof/argument systems have a so-called *unbalanced* property, where the verifier's complexity is minimized (sometimes maybe at the cost of increasing the prover's complexity). This property is desirable when the number of verifiers is large, such as the blockchain scenarios.

3 Security Definition

In this section, we formally define the scriptable NIZK. Our definition is through an ideal functionality $\mathcal{F}^{\mathsf{Q}}_{\text{sNIZK}}$. In addition, we present a setup functionality $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$. We note that the two functionalities will be realized in section 4 and instantiated in section 5, respectively.

Functionality $\mathcal{F}^{\mathsf{Q}}_{\text{sNIZK}}$

The functionality interacts with a set of parties $\mathcal{P} := \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$. It is parameterized with a predicate Q.

Proof: Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}\rangle)$ from a party $P_i \in \mathcal{P}$:

- Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output}$;
- Send $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle P_i, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}\rangle)$ to $\mathcal{S}$;
- Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \pi)$ from $\mathcal{S}$, record tuple $(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi)$ and return $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi\rangle)$ to P_i.

Verification: Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi\rangle)$ from a party $P_j \in \mathcal{P}$:

- If tuple $(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi)$ is not recorded, send $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle P_j, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi\rangle)$ to $\mathcal{S}$;
- Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \mathsf{Input}_{\mathrm{priv}})$ from $\mathcal{S}$:
  - Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output}$;
  - Record the tuple $(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi)$;
- If a tuple $(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi)$ has been recorded, return $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, 1)$; else, return $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, 0)$.

Fig. 1: The scriptable functionality $\mathcal{F}^{\mathsf{Q}}_{\text{sNIZK}}$.

Scriptable NIZK ideal functionality. The scriptable NIZK ideal functionality $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ is depicted in Fig. 1. The functionality is parameterized by a predicate Q. Given a script $\mathcal{C}$, a public input $\mathsf{Input}_{\text{pub}}$, a private input $\mathsf{Input}_{\text{priv}}$, and an output Output, the functionality $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ allows the prover to obtain a proof π if $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}) = \mathsf{Output}$. Once a proof π is generated, it will always is verified. Notice that the proof π is generated without the knowledge of the private input $\mathsf{Input}_{\text{priv}}$; therefore, the proof generated by $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ has the conventional zero-knowledge. Since $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ must obtain a private input $\mathsf{Input}_{\text{priv}}$ such that $\mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}) = \mathsf{Output}$ before recording a proof π. Hence, $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ also capture the (knowledge) soundness property. In addition, the scriptable property is reflected by the predicate Q, which restricts the class of functions that $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ supports. For instance, Q could be the total execution steps is less than a certain bound.

The functionality $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ interacts with a set of players $\mathcal{P} := \{P_1, \ldots, P_n\}$ as well as ideal adversary $\mathcal{S}$. To generate a proof π, the prover needs to submit the command $\langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, \mathsf{Output}\rangle$ to $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$. After checking the validity, $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ will inform the adversary $\mathcal{S}$ using command $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, P_i, \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Output})$. If the adversary $\mathcal{S}$ allows, she will then send the proof π to $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$. $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ records the message $(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Output}, \pi)$ and returns it to the requestor. To verify a proof π, the functionality $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ first checks if the tuple $(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Output}, \pi)$ is recorded. If not, which means the proof is not generated by the functionality itself, then $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ asks the adversary $\mathcal{S}$ for the private input. Once a private input $\mathsf{Input}_{\text{priv}}$ is submitted, $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ checks $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}) = \mathsf{Output}$. If it is the case, $\mathcal{F}_{\text{sNIZK}}^{\mathsf{Q}}$ records the tuple $(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Output}, \pi)$, and the proof is accepted.

Remark on succinctness. We say a NIZK proof system is succinct if the size of the proof $|\pi| = \text{poly}(\lambda)(|x| + |w|)^{o(1)}$.

Functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$

The functionality interacts with a set of parties $\mathcal{P} := \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$. It is parameterized with a predicate Q and a digital signature scheme $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$.

- Upon receiving $(\textsc{Init}, \mathsf{sid})$ for the first time from any party $P_i \in \mathcal{P}$:
  - Generate $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$;
  - Record $(\mathsf{sid}, \mathsf{PK}, \mathsf{SK})$;
- Upon receiving $(\textsc{GetPK}, \mathsf{sid})$ from a party $P_i \in \mathcal{P}$:
  - If $(\mathsf{sid}, \mathsf{PK}, \cdot)$ is recorded, return $(\textsc{GetPK}, \mathsf{sid}, \mathsf{PK})$ to the requestor P_i.
- Upon receiving $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}\rangle)$ from a party $P_i \in \mathcal{P}$ for some ssid, if $(\mathsf{sid}, \cdot, \mathsf{SK})$ is recorded, send $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle P_i, \mathcal{C}, \mathsf{Input}_{\text{pub}}\rangle)$ to the adversary $\mathcal{S}$; Once receiving $(\textsc{Proceed}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$, do:
  - Execute $y \leftarrow \mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$;
  - Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, y) = 1$;
  - Sign $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\text{pub}}, y\rangle)$;
  - Return $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle y, \sigma\rangle)$ to the requestor P_i.

Fig. 2: The Q-compliant trusted hardware functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$.

Q**-compliant trusted hardware model.** Our scheme is built in the Q-*compliant trusted hardware model* (Q-HW model), where Q is a predicate that specifies the class of functions that the hardware is allowed to compute. In the Q-HW model, all parties have access to an ideal functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$, which on input queries, executes a given Q-compliant function and returns the execution results. The predicate Q depends on the setup, which may vary from protocol to protocol. In this work, we abstract our requirement as the functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$ (cf. Fig. 2, below). The $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$ functionality is parameterized with a predicate Q and a digital signature scheme, denoted $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$ can be initialized once by sending the $(\textsc{Init}, \mathsf{sid})$ command to it. It then generates $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$ and record $(\mathsf{sid}, \mathsf{PK}, \mathsf{SK})$. After initialization, anyone can query the public key PK using the $\textsc{GetPK}$ command. Anyone can then send $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$ request to the functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Q}}$, where $\mathcal{C}$ is the polynomial-time algorithm, $\mathsf{Input}_{\text{pub}}$ is the public input, and $\mathsf{Input}_{\text{priv}}$ is the private input. The functionality first computes $\mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}) = y$ and then asserts $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, y) = 1$; it then returns (y, σ), where the signature $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\text{pub}}, y\rangle)$. Note that the private input is not signed.

4 Our Succinct Scriptable NIZK Construction

In this section, we present our succinct scriptable NIZK construction in the $\mathcal{O}_{\text{HW}}$-hybrid world. Before presenting our intuition and construction, we first set up the context for succinct scriptable NIZK.

Common information. Unlike most existing NIZK proof systems, the script $\mathcal{C}$ (or language $\mathcal{L}$ to be proven) is not hardcoded in the prover and verifier executable files. Our script NIZK proof system allows the users to configure the language instance. This implicitly assumes that the prover and the verifier(s) have some common information in addition to the statement x before the protocol execution. For instance, they all know the description of the NP language $\mathcal{L}$, which is usually represented by its polynomially decidable binary relation $\mathcal{R}$. Without loss of generality, for a relation $\mathcal{R}$, we assume there exists an efficiently computable algorithm $\mathcal{C}_{\mathcal{R}}$ such that $\mathcal{C}_{\mathcal{R}}(x, w) = 1$ if $(x, w) \in \mathcal{R}$ and otherwise $\mathcal{C}_{\mathcal{R}}(x, w) = 0$. $\mathcal{C}_{\mathcal{R}}$ is the common public input to both the prover and the verifier. Depending on the concrete implementation, different NIZK proof systems use different $\mathcal{C}_{\mathcal{R}}$ representation; most popular NIZK proof systems use arithmetic circuit representation, while some, e.g. [5], allows more developer-friendly representations, e.g., in C programming language. Although, in principle, one can convert any RAM model program into a circuit representation, this transform imposes $O(\log n)$ overhead.

Intuition. Trusted hardware offers two important features: (i) data confidentiality and (ii) computation integrity. Most existing trusted hardware (TEE) based applications, e.g., [15] mainly explore the data confidentiality aspect; whereas, in this project, we emphasize the computation integrity aspect. Recall that in a NIZK proof, the prover and the verifier have common input $(\mathcal{C}_{\mathcal{R}}, \mathsf{Input}_{\text{pub}} := x)$. The potentially malicious prover wants to convince the verifiers that he/she knows a witness $\mathsf{Input}_{\text{priv}} := w$ such that $\mathcal{C}_{\mathcal{R}}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}) = 1$. Since the trusted hardware can guarantee computation integrity even when the host is malicious, we can let $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ to execute the relationship decision algorithm $b \leftarrow \mathcal{C}_{\mathcal{R}}(x, w)$ and sign the output b. To bind the decision algorithm and statement, we let $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ signs $(\mathcal{C}_{\mathcal{R}}, x, b)$ without revealing the witness w. Therefore, by checking the signature, the verifier is convinced that the prover must know a witness w such that $\mathcal{C}_{\mathcal{R}}(x, w) = 1$ if $(\mathcal{C}_{\mathcal{R}}, x, 1)$ is signed by $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$. Similarly, for general computation, the private input $\mathsf{Input}_{\mathrm{priv}}$ is not signed; therefore, zero-knowledge property is preserved even if the signature leaks the signed message.

What is the difference between the above NIZK construction and trusted computation in the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ functionality setting? Recall that NIZK proofs are typically deployed in a one-to-many scenario, so the prover only needs to invoke the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ once and many verifiers can check the validity of the proof; on the contrary, the other existing TEE based trusted computation applications mostly focus on one-to-one setting. Our crs is just the public key of $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$.

Construction. Our Succinct Scriptable NIZK construction utilizes the Q-compliant hardware functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ as defined in Fig. 2.

Succinct scriptable NIZK protocol $\Pi_{\mathrm{NIZK}}^{\mathsf{Q}}$

Proof: Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output} \rangle)$ from the environment $\mathcal{Z}$, $P_i \in \mathcal{P}$ does:

- *If the functionality* $\mathcal{O}_{HW}^{\mathsf{Q}}$ *is not initialized yet, send* $(\textsc{Init}, \mathsf{sid})$ *to* $\mathcal{O}_{HW}^{\mathsf{Q}}$*;*
- Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output}$;
- Send query $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ to $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ and obtain $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathsf{Output}, \sigma \rangle)$ from $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$;
- Output $(\textsc{ProveReturn}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to the environment $\mathcal{Z}$.

Verification: Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}, \pi \rangle)$ from the environment $\mathcal{Z}$, $P_j \in \mathcal{P}$ does:

- *Query* $(\textsc{GetPK}, \mathsf{sid})$ *to* $\mathcal{O}_{HW}^{\mathsf{Q}}$*, obtaining* $(\textsc{GetPK}, \mathsf{sid}, \mathsf{PK})$*;*
- Parse π as σ;
- Compute $b \leftarrow \mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output} \rangle, \sigma)$;
- Output $(\textsc{VerifyReturn}, \mathsf{sid}, \mathsf{ssid}, b)$ to the environment $\mathcal{Z}$.

Fig. 3: The succinct scriptable NIZK protocol $\Pi_{\mathrm{NIZK}}^{\mathsf{Q}}$ in the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$-hybrid model.

We aim to achieve constant verification time; light-weight device can perform the verification. In addition, the verifier is only required to query the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ functionality once to obtain the public key PK; when PK has already been fetched, the verification can be executed offline. As depicted in Fig. 3, our succinct scriptable NIZK proof protocol $\Pi_{\mathrm{NIZK}}^{\mathsf{Q}}$ uses a digital signature scheme $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ as its building block. At the beginning of the protocol, the hardware functionality needs to be initialized. In Fig. 3, this step is performed by the prover (marked in grey) if it is not done yet. The prover then asserts $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output}$; it sends $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ to $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ and obtains $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathsf{Output}, \sigma \rangle)$ from $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$. σ is the proof.

To verify a proof π, the verifier needs to know the public key PK. This step can be performed by a trusted setup, and PK is published as the common reference string. Otherwise, the verifier can query $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ to fetch it (marked in grey). In the verification phase, the verifier V accepts the proof if $\mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}\rangle, \sigma) = 1$.

Security. We show the security of our succinct scriptable NIZK construction via Thm. 1, below. Its proof can be found in the full version.

Theorem 1. *Assume signature scheme* $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is EUF-CMA secure. The scriptable NIZK protocol* Π^{Q}_{NIZK} *described in Fig. 3, UC-realizes the* $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sNIZK}}$ *functionality depicted in Fig. 1 in the* $\mathcal{O}^{\mathsf{Q}}_{HW}$*-hybrid world.*

5 $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ Instantiations

In this section, we realize the Q-compliant trusted hardware functionality $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ via Intel SGX and Arm TrustZone.

Challenges. In both platforms, there are a number of challenges need to be resolved. In terms of SGX, the remote attestation of Intel SGX currently requires the verifier to contact the Intel IAS server. On the other hand, in a typical NIZK proof system usage case, the prover aims to prove the truth of the statement to a great number of verifiers. If each verifier needs to query the Intel IAS server to check the proof, the overall performance is limited by Intel's throughput. Moreover, the validity of a NIZK proof should be consistent over time, i.e., if a NIZK proof is verifiable at this moment, the same proof should remain verifiable in the future. Unfortunately, this would not be the case if we invoke the Intel IAS in the verification process; certifying an old quote (say, generated 1 year ago) is never the design goal of Intel's remote attestation. This is because the quote needs to contain an *non-revoked proof* for each item on the signature revocation list, and the proof is no longer verifiable once the revocation list is updated at the Intel side. That means a quote is only valid until the next revocation list update. To resolve this issue, in our design, after generating the quote, the prover immediately queries the Intel IAS server for the attestation verification report on behave of a verifier. Since the attestation verification report is signed by Intel, given Intel's public key, anyone can verify the validity of the attached signature. This tweak also makes the verification process non-interactive.

Secondly, the existing SGX-based proof system, e.g., [40], requires the prover and the verifiers agree on the executable binary (enclave) for the language to be proven. It would make it impossible to build a universal NIZK system in practice. Note that SGX only signs the measure of the enclave, which cannot be directly compared with the corresponding algorithm. Imaging a verifier who is checking a NIZK proof generated some time ago, how would the verifier know the executable binary (enclave) is faithfully compiled? Therefore, NIZK systems, like [40], would need a trusted party to generate an executable binary (enclave) for a given problem instance, and the binary is served as the concrete CRS for the given instance.

In terms of TrustZone, unlike the ecosystem of SGX that is controlled by Intel, the fragmentation of the ARM TrustZone ecosystem may make it hard to have a unique setup standard. To resolve this issue, we need to introduce a trusted setup authority to serve as an attestation server.

SGX-based system overview. In our system, the protocol Π_{SGX} involves three entities: the (trusted) Intel server, denoted as IS, the prover P, and the SGX hardware, denoted as HW_{SGX}. In practice, it is still a challenge for a third party to verify the consistency between an executable binary and its software specification. That is, the binary contains no bug, no trapdoor, and it is not subverted. Even it is possible, it dramatically increases the verifier's complexity. On the other hand, it is implausible to assume a trusted third party that is available to generate a certified binary for each problem instance. To address this issue, we decide to adopt a scripting language, called Lua. Lua is a lightweight script language, which is ideal for the SGX enclave computation environment. We let a trusted party, i.e., the (trusted) Intel server IS, to produce a Lua script engine enclave $\mathcal{SE}$. IS then signs $\mathcal{SE}$ so that no one can tamper with its functionality. As depicted in Fig. 4, $\mathcal{SE}$ has one main function called $\mathsf{VerifySign}$[4]. It takes three arguments: (i) a script $\mathcal{C}$, (ii) a public input $\mathsf{Input}_{\text{pub}}$ (iii) a tag, tag, that can be used to specify the proof context, such as $ssid$, etc. The $\mathsf{VerifySign}$ function first loads the private input $\mathsf{Input}_{\text{priv}}$ from the prover; it then executes the script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$ using the script interpreter. Abort if $y = \perp$, which means the execution error happened; that is considered as $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, y) = 0$. Otherwise, it sets $h := \mathsf{hash}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y)$ and $\mathsf{ReportData} := (\mathsf{tag}, h)$; it then invokes $\mathsf{EREPORT}$ to create a report r for QE to sign. Finally, it returns (y, r).

```
Enclave $\mathcal{SE}$

$\mathsf{VerifySign}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{tag})$ :
  – (OCALL) Load $\mathsf{Input}_{\text{priv}}$;
  – Execute script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$;
  – Set $\mathsf{ReportData} = (\mathsf{tag}, \mathsf{hash}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y))$;
  – (EREPORT) Create report $r$ for QE to sign;
  – Return $(y, r)$;
```

Fig. 4: The script engine enclave $\mathcal{SE}$.

Remark. Technically, the private input $\mathsf{Input}_{\text{priv}}$ can be input to the $\mathsf{VerifySign}$ function together with the script $\mathcal{C}$ and the public input $\mathsf{Input}_{\text{pub}}$ as another argument. We choose to load $\mathsf{Input}_{\text{priv}}$ separately during the enclave execution for the sake of uniformity: (i) for some applications, we could choose to hard code $\mathcal{C}$ and $\mathsf{Input}_{\text{pub}}$ for efficiency; and (ii) in case that the prover needs to use an SGX enabled server from a third party, it is possible to load $\mathsf{Input}_{\text{priv}}$ in to the enclave via secure channels to ensure privacy.

The hardware functionality $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ is instantiated by the protocol $\Pi^{\mathsf{Q}}_{\text{SGX}}$ shown in Fig. 5. The $\textsc{Init}$ functionality is realized by the Intel server IS and the hardware HW_{SGX}. Upon receiving $(\textsc{Init}, sid)$, IS invokes the EPID provisioning key procedure [26] with HW_{SGX}. The root seal key of HW_{SGX} was generated during the processor manufacturing, and Intel claims that they are oblivious to it; the root provisioning key is set up by a special purpose offline key generation facility. The actual procedure is complicated; HW_{SGX} is registered to the Intel server IS via a blind joining protocol. We refer interested reader to [26] for details. Hereby, we simplify the description – at the end, HW_{SGX} stores a group signature secret key GSK, and the Intel server IS stores the corresponding group signature public key GPK that allows it to verify the signatures generated

[4] The enclave also has a GetQEInfo function to receive the target information of QE. It is omitted for simplicity.

Protocol $\mathit{\Pi}^{\mathsf{Q}}_{\text{SGX}}$

Init: Upon receiving ($\textsc{Init}$, sid), the Intel server IS interacts with HW_{SGX} invoking the EPID provisioning key procedure (Cf. [26]); At the end of the protocol:

- The Intel server IS stores GPK;
- HW_{SGX} stores GSK;

The Intel server IS also does:

- Generate $(\widetilde{\mathsf{PK}}, \widetilde{\mathsf{SK}}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$;
- Create the script engine enclave $\mathcal{SE}$ as depicted in Fig. 4;
- Sign $\tilde{\sigma} \leftarrow \mathsf{DS.Sign}(\widetilde{\mathsf{SK}}, \mathcal{SE})$;

GetPK: Upon receiving ($\textsc{GetPK}$, sid), the Intel server IS sets $\mathsf{PK}^* := (\widetilde{\mathsf{PK}}, \mathcal{SE}, \tilde{\sigma})$ and return ($\textsc{GetPK}$, sid, PK^*);

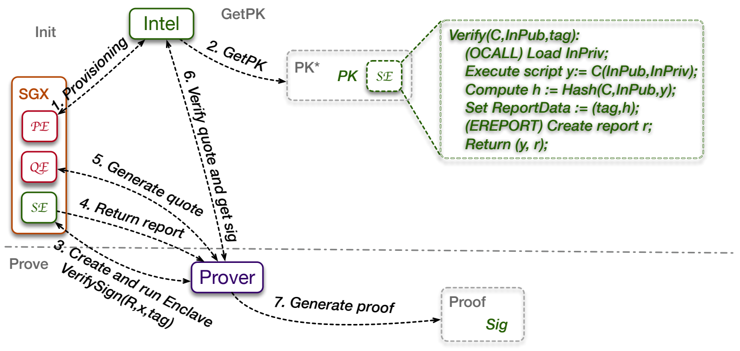
Prove: Upon receiving ($\textsc{Compute}$, sid, $ssid$, $\langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}} \rangle$):

- The prover P_i creates an enclave instance of $\mathcal{SE}$ to HW_{SGX};
- The prover P_i invokes $\mathsf{VerifySign}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{tag} := (sid, ssid))$;
  (Supply $\mathsf{Input}_{\text{priv}}$ during the execution);
- HW_{SGX} runs $y \leftarrow \mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$ and aborts if $y = \perp$ (i.e.
  $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, y) = 0$);
  Otherwise, it outputs a quote $q(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag})$;
- The prover P_i sends the quote $q(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag})$ to the Intel server IS to verify.
- The Intel server IS checks the validity of the quote; it then signs and returns
  $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag} \rangle)$;
- The prover P_i outputs (y, σ);

Fig. 5: Protocol $\mathit{\Pi}^{\mathsf{Q}}_{\text{SGX}}$ realizing $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ via Intel SGX.

by HW_{SGX}. Note that the group signature is only used to authenticate HW_{SGX} to the Intel, rather than to the public. Therefore, it is possible to replace the group signature scheme with some symmetric key cryptographic primitive, e.g., MAC. In addition, IS also generates $(\widetilde{\mathsf{PK}}, \widetilde{\mathsf{SK}}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$. It then creates the script engine enclave $\mathcal{SE}$ as depicted in Fig. 4 and signs it $\tilde{\sigma} \leftarrow \mathsf{DS.Sign}(\widetilde{\mathsf{SK}}, \mathcal{SE})$. The public key is defined as $\mathsf{PK}^* := (\widetilde{\mathsf{PK}}, \mathcal{SE}, \tilde{\sigma})$. Anyone can query ($\textsc{GetPK}$, sid) to the Intel server IS to fetch the public key PK^*. The $\textsc{Compute}$ command is realized by all three parties. Upon receiving ($\textsc{Compute}$, sid, $ssid$, $\langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}} \rangle$), the prover P_i creates an enclave instance of $\mathcal{SE}$ to HW_{SGX}; it then invokes $\mathsf{VerifySign}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{tag})$ (supplying $\mathsf{Input}_{\text{priv}}$ during the execution). HW_{SGX} executes the script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}})$; Abort, if $y = \perp$, which is considered as $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}, y) = 0$. Otherwise, it outputs a report $r(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag})$ for local attestation. The prover P_i sends the report $r(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag})$ to the QE of HW_{SGX} to produce a quote $q(\mathcal{C}(\mathsf{Input}_{\text{pub}}, \mathsf{Input}_{\text{priv}}))$; the prover P_i sends the quote $q(\mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag})$ to the Intel server IS to verify. The above steps are simplified in Fig. 5. The Intel server IS checks the validity of the quote, i.e., checking the group signature and that the SGX platform generating the quote is not revoked; it then signs and returns $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathcal{C}, \mathsf{Input}_{\text{pub}}, y, \mathsf{tag} \rangle)$; The prover P_i outputs (b, σ); Fig. 6 summaries the basic flow for the $\textsc{Init}$, $\textsc{GetPK}$, and $\textsc{Compute}$ protocols.

TrustZone-based system overview. ARM TrustZone is another popular trusted hardware platform that can also be leveraged (as long as a device-unique, asymmetric key pair signed by the device's vendor exists). ARM TrustZone provides isolated exe-



```
Verify(C,InPub,tag):
    (OCALL) Load InPriv;
    Execute script y:= C(InPub,InPriv);
    Compute h := Hash(C,InPub,y);
    Set ReportData := (tag,h);
    (EREPORT) Create report r;
    Return (y, r);
```

Fig. 6: SGX based trusted hardware instantiation

cution by separating the CPU into two different worlds, i.e., normal world and secure world. The code running inside the normal world cannot directly access the resource inside the secure world. Also only the application inside the secure world can access the protected resource.

Specifically, the device-unique key pair can be used to sign the attention blob that indicates the attestation data originates from the secure world. The attestation data in this case contains $\langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, y, \mathsf{tag} \rangle$. The signed data will be passed to the attestation server of device vendor (like Intel IAS). If the signature verification passes on the device vendor's attestation server, the prover generates proof.

The Lua script engine design and system architecture is similar to the SGX-based solution. However, it is more efficient, as the attestation data can be verified without interacting with the the attestation server if the verifier already fetched the public key PK from it.

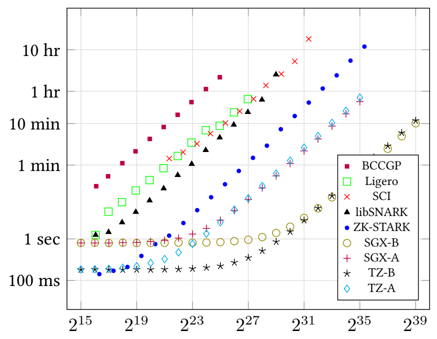
6 Implementation and Evaluations

Our SGX-based prototype is implemented in C++ using the Intel(R) SGX SDK v2.5 for Linux. Our implementation is built on top of [36], and we added OpenSSL lib functions for common cryptographic primitives, such as SHA256, ECDSA, etc. Since system call is not allowed in enclave, we also simulated a simple file system to support the Lua interpreter. The size of the compiled enclave binary is approximately 3.2 MB. In appendix A, we will present more detail on our SGX-based prototype.

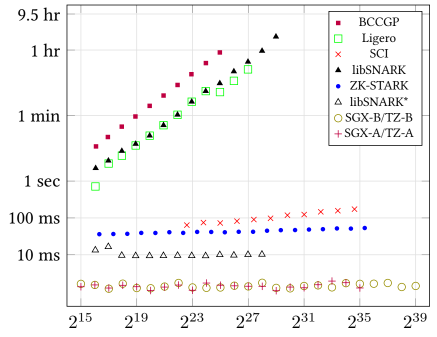
Our TrustZone-based prototype is developed on the Hikey 960 development board, which is powered by Huawei Kirin 960 SoC with 4 ARM Cortex-A73 cores and 4 1.8GHz ARM Cortex-A53 cores. There are 4GB DDR4 memory and 32GB UFS flash on our board. In our experiment, we choose OPTEE (v3.6) as the OS in the secure world, which is open source and well maintained. For the normal world OS, we use a Linux distribution, which is developed by Linaro Security Working Group based on Linux kernel v5.1 and able to corporate with OPTEE. Then, we implement a Trusted App (TA) for the secure world, which will be managed by OPTEE. The Client Application (CA) in the normal world can invoke the TA through specific interface. Lua Intrepreter (v5.3.2) is adopted and modified. The default secure memory size supported by OPTEE is 16 MB, which restricts the script size. A signing key is stored in the TrustZone for the experiment. The enclave structure and system design is similar to the SGX-based solution, except we adopt ECDSA signature over the `secp256k1` curve. Therefore, the signature/proof size is only 32 Bytes.

Fig. 7a,7b,7c shows the performance comparison of different succinct NIZK proof systems w.r.t. prover's running time, verifier's running time, and proof size, respectively. The complexity is measured by the number of multiplication gates. Our work and BCCGP are 128bit security; libSNARK and SCI are 80-bit security; Ligero and zk-STARK are 60-bit security. Our system is tested on a SGX-equipped processor (i7-8700 @ 3.2GHz and 16GB RAM, single thread) and Hikey 960 TrustZone development board. All the other systems were tested on a server with 32 AMD cores @ 3.2GHz and 512GB RAM, and the data was reported by [3]. For libSNARK, the hollow marks (libSNARK*) in verifier time and proof size measure only count the post processing phase; while solid marks also count CRS generation time. For our SGX based scheme, the prover's running time includes network time for Intel IAS verification; SGX-A (TZ-A) stands for arithmetic circuit over ring $\mathbb{Z}_{2^{64}}$, and SGX-B (TZ-B) stands for Boolean circuit (NAND gates) w.r.t. SGX and TrustZone platforms.

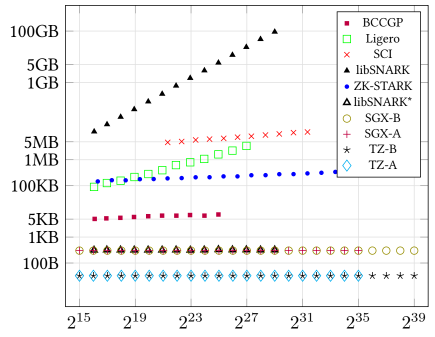
Although our NIZK proof system support RAM model computer program, we implemented circuit evaluation as Lua script to facilitate comparison. We emphasize that the reported time is tested using Lua scripts. If the circuit is written in native C, the performance is approximate 10 times better on both SGX and TrustZone platforms. The complexity is measured by the number of multiplication gates. We provide 'SGX-A' and 'TZ-A' as the benchmark for



(a) Prover Time



(b) Verifier Time



(c) Proof Size

Fig. 7: Performance comparison of different succinct NIZKs.

arithmetic circuit over ring $\mathbb{Z}_{2^{64}}$ for SGX and TrustZone, respectively; 'SGX-B' and 'TZ-B' as the benchmark for Boolean circuit, using SIMD to implement NAND gates. The measure of the enclave is assumed to be pre-computed and announce by Intel, so it is not counted into the verifier's running time; moreover, the problem instance consists of the Lua script and its hash; otherwise, the verifier can also compute the hash at a small cost. As shown in [14], SHA256 can be performed at 2.1-3.5GB/s on most platforms.

7 Related work

Universal NIZK. Now we briefly describe several different practical approaches for universal NIZK (i.e., can be applied to general computations and languages in NP). We note that our description here are based on a large body of existing results, and unfortunately we cannot cover the entire body research in this line. We mainly compare the performance related properties, including prover scalability, verifier scalability, setup/initialization scalability, and communication scalability. Additionally, we also compare the underlying setup assumptions and computational assumptions. We note that, in the existing approaches, each setup only support one language instance. Meanwhile, our scriptable NIZK can support multiple language instances in a single setup.

There are multiple approaches to scalable NIZK. The *first approach* is based on homomorphic public-key cryptography, by Ishai et al. [24] and Groth [21]. Then Gennaro et. al [16] introduced an extremely efficient instantiation, based on Quadratic Span Programs, which later been implemented in Pinocchio [35]; see also [5, 6, 12, 27]. Note that, this technique has been used in Zcash. We note that, the homomorphic public-key cryptography based approach can be combined with other techniques to improve the performance. For example, Valiant, [42] suggested to reduce prover space consumption via knowledge extraction assumptions; This combined method can inherit most of the properties from the underlying proof system. We note that our scriptable NIZK system is more efficient.

The *second approach* is based on the hardness of the DLP, originally proposed by Groth [22] and then implemented in [11, 8]. [11] Note that, the communication complexity in the DLP approach is logarithmic. However, the verifier complexity in this approach is not scalable. The *third approach* is based on efficient Interactive Proofs (IP) [20, 37]. The line of realizations can be found in [44] and [43]. Note that, the verifier in this approach is not scalable. The *fourth approach* is via the so-called "MPC in the head", originally suggested by Ishai et al. [25] and then implemented in ZKBoo [18], and in Ligero [1]. "MPC in the head" based systems have a non-scalable verifier; in addition, communication complexity is non-scalable. A recent proposal called STARK [4], attempts to simultaneously minimize proof size and verifier computation. However, their proof sizes are not small.

In [30, 23], an updatable and universal reference string is used. The main goals of this approach is to address risks surrounding setups and many other security challenges in practice. It does not improve the efficiency. Another method to achieve universal setup is using universal circuit [41, 28]. In [6, 5], a TinyRAM architecture is used to describe universal computations as simple programs. A universal circuit is built based on a specific universal language (i.e., a set of tuples, where each tuple consists of a TinyRAM program, an input string, and a time-bound to run the program). Unfortunately, this approach incur a large overhead on the prover computation.

Trusted hardware. Many previous works have proposed using trusted hardware to build cryptographic algorithms and systems, including protection of cryptographic keys [31], functional encryption [15], digital rights management [39], map-reduce jobs [32, 13], machine learning [33], data analytics [38], and protecting unmodified Windows applications [2].

8 Conclusion

In this work, we introduce a new notion called succinct script NIZK proof system. We formally model this notion in the UC framework. We then propose a generic scriptable NIZK solution based on trusted hardware. We also instantiated our scheme in both Intel SGX and Arm TrustZone. To the best of our knowledge, the proposed succinct scriptable NIZK is better than all the existing succinct NIZK proof systems w.r.t. the prover running time (1000 times faster for Lua script, 10000 times faster for Native C), the verifier's running time (10 times faster), and the proof size (10 times smaller). Most importantly, our NIZK proof system can be readily deployed and used by any developers without the need of cryptographic background.

Acknowledgement

Bingsheng Zhang is supported by the Key (Keygrant) Project of Chinese Ministry of Education. (No. 2020KJ010201) and the National Natural Science Foundation of China (Grant No. 62072401). Hong-Sheng Zhou and Phuc Thai are supported by NSF grant CNS-1801470, a Google Faculty Research Award and a research gift from Ergo Platform. Yajin Zhou and Kui Ren are also supported by the Open Project Program of Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province. Yajin Zhou is the corresponding author.

References

1. Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. *ACM CCS 2017*, pages 2087–2104.
2. Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.
3. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.
4. Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

5. Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108.
6. Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. *USENIX Security 2014*, pages 781–796.
7. Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *20th ACM STOC*, pages 103–112.
8. Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357.
9. E. Brickell and J. Li. Enhanced privacy id from bilinear pairing for hardware authentication and attestation. In *2010 IEEE Second International Conference on Social Computing*, pages 768–775.
10. Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.
11. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334.
12. George Danezis, Cédric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct NIZK arguments. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 532–550.
13. Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling stronger privacy in MapReduce computation. *USENIX Security 2015*, pages 447–462.
14. ECRYPT. ebacs: Ecrypt benchmarking of cryptographic systems. https://bench.cr.yp.to/results-hash.html, 2018. Last accessed: 2019-05-11.
15. Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using intel SGX. *ACM CCS 2017*, pages 765–782.
16. Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645.
17. Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. *43rd ACM STOC*, pages 99–108.
18. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. ZKBoo: Faster zero-knowledge for boolean circuits. Cryptology ePrint Archive, Report 2016/163, 2016. http://eprint.iacr.org/2016/163.
19. Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994.
20. Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. *40th ACM STOC*, pages 113–122.
21. Jens Groth. Fully anonymous group signatures without random oracles. *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 164–180.
22. Jens Groth. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 431–448.
23. Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728.

24. Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short pcps. In *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC'07)*, pages 278–291, June 2007.
25. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. *39th ACM STOC*, pages 21–30.
26. Simon P Johnson, Vincent R Scarlata, Carlos V Rozas, Ernie Brickell, and Frank McKeen. Intel sgx: Epid provisioning and attestation services. *Intel*, 2016.
27. SCIPR Lab. libsnark: a c++ library for zksnark proofs, 2019.
28. Helger Lipmaa, Payman Mohassel, and Saeed Sadeghian. Valiant's universal circuit: Improvements, implementation, and applications. Cryptology ePrint Archive, Report 2016/017, 2016. https://eprint.iacr.org/2016/017.
29. Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. *USENIX Security 2016*, pages 549–564.
30. Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updateable structured reference strings. *IACR Cryptology ePrint Archive*, 2019:99, 2019.
31. Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.
32. Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. *ACM CCS 2015*, pages 1570–1581.
33. Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. *USENIX Security 2016*, pages 619–636.
34. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252,
35. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252.
36. Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica, and Marcelo Pasin. A lightweight mapreduce framework for secure processing with sgx. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '17, pages 1100–1107.
37. Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. *48th ACM STOC*, pages 49–62.
38. Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54.
39. G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 357–368. ACM, 2014.
40. F. Tramer, F. Zhang, H. Lin, J. Hubaux, A. Juels, and E. Shi. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *Euro S&P 2017*, pages 19–34, 2017.
41. Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, pages 196–203.
42. Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. *TCC 2008*, volume 4948 of *LNCS*, pages 1–18.
43. Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943.

44. Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy*, pages 863–880.

A SGX implementation

As we mentioned in Section 6, our SGX-based prototype is implemented in C++ using the Intel(R) SGX SDK v2.5 for Linux. Our implementation is built on top of [36], and we added OpenSSL lib functions for common cryptographic primitives, such as SHA256, ECDSA, etc. Since system call is not allowed in enclave, we also simulated a simple file system to support the Lua interpreter. The size of the compiled enclave binary is approximately 3.2 MB.

Up on execution, the prover first creates an instance of the Lua script engine enclave in the SGX and transfers the target information of QE into the Lua script engine enclave, which will be used later to generate the report for QE. The prover then produces his proof by calling specific function interface of the enclave, VerifySign, taking the script $\mathcal{C}$ and the public input $\mathsf{Input}_{\mathrm{pub}}$ as the arguments of the function. In our prototype, the script $\mathcal{C}$ and statement $\mathsf{Input}_{\mathrm{pub}}$ are pre-loaded into the simulated filesystem. After loading $\mathsf{Input}_{\mathrm{priv}}$ from the prover and putting it into the simulated filesystem, the enclave invokes the Lua interpreter to process the script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$, where the script can access the statement and witness through Lua file operations. Note that Lua heap size need to be predefined while compiling the Lua script engine enclave, such as 32 MB, which restrict the class of script it can support.

After the script execution, the enclave hashes $h := \mathsf{hash}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ and then put (tag, h) in to the REPORTDATA field of the report structure, and generate the report $r(\mathsf{tag}, h)$ for QE to sign. The prover will then fetch the report $r(\mathsf{tag}, h)$ and send it together with signature revocation list (which can be obtained from the Intel IAS and SPID (which is assigned by the Intel IAS when user registers to the Intel IAS) to the QE. The QE will verify the report using its report key and compute an non-revoked proof for the signature revocation list, generating a quote consisting of the ReportBody field of the report, the non-revoke proof and some other necessary information. The prover then will send the quote to the Intel IAS server for attestation verification report.

Reducing proof size. Naively, the prover can send the entire signed attestation verification report as the NIZK proof. The proof size is 731 Bytes (IAS report size) $+$ 256 Bytes (the signature size). To reduce proof size, we observe that Intel's signature is signed on top of the hash of the attestation verification report, so the prover does not need to give the entire report as a part of the proof as far as the verifier can reproduce the hash of the report. However, the verifier is interested in some field of in the isvEnclaveQuoteBody, such as REPORTDATA. Notice that SHA256 uses Merkle-Damgård structure, i.e., the final hash digest is calculated by iteratively calling a compression function over trunks of the signing document. Therefore, the prover can give the partial hash digest of the first part of the signing report, including ID, timestamp,

Table 2: QuoteBody Structure

uint16_t	version;
uint16_t	sign_type;
sgx_epid_id_t	epid_group_id;
sgx_isv_svn_t	qe_svn;
sgx_isv_svn_t	pce_svn;
uint32_t	xeid;
sgx_basename_t	basename;
sgx_cpu_svn_t	cpu_svn;
sgx_mise_select_t	misc_select;
uint8_t	reserved1[28];
sgx_attributes_t	attributes;
sgx_measurement_t	mr_enclave;
uint8_t	reserved2[32];
sgx_measurement_t	mr_signer;
uint8_t	reserved3[96];
sgx_prod_id_t	isv_prod_id;
sgx_isv_svn_t	isv_svn;
uint8_t	reserved4[60];
sgx_report_data_t	report_data;

version, isvEnclaveQuoteStatus. The isvEnclaveQuoteBody structure is shown in Table 2. The verifier is only interested in the five fields marked in grey background, and they can be reconstructed from the public input of the verifier. Moreover, currently, all the reserved fields must be 0. Moreover, the verifier also wants to check isvEnclaveQuoteStatus $=$ OK; nevertheless, we observe that the attestation verification report whose isvEnclaveQuoteStatus $=$ OK has a fixed length n. Otherwise, the length of the attestation verification report is different from n. Based on that observation, we can regard the length n as another public input of the verifier. Then when the verifier receives a proof, he/she can check whether the isvEnclaveQuoteStatus field of the associated attestation verification report is OK by putting the length n into the end of the report as the total hashed length. then if the isvEnclaveQuoteStatus field is not OK, the report hash is not aligned probably, resulting a wrong hash digest.

We let the prover give the partial hash digest until misc_select field. Denote the partial hash digest of the report as ph. The prover needs to provide the attributes field, denoted as attr, which is 16 Bytes[5]. The proof is $(ph, \mathsf{attr}, \sigma)$. The verifier can use reconstruct the hash of the report and then check the validity of the signature. The proof size is now reduced to 41 Bytes + 256 Bytes (the signature size), which is 297 Bytes.

[5] In fact, there are 56 bits reserved area, whose default value is 0 in the attributes field. Hence, the size can be further reduced by 56 bits.