Scriptable and Composable SNARKs in the Trusted Hardware Model[1]

Zhelei Zhou [a], Bingsheng Zhang [a,*], Yuan Chen[a], Jiaqi Li[a], Yajin Zhou [a], Yibiao Lu[a], Kui Ren [a], Phuc Thai [b] and Hong-Sheng Zhou [b]
[a] *School of Computer Science and Technology, Zhejiang University, Zhejiang, China*
E-mails: zl_zhou@zju.edu.cn, bingsheng@zju.edu.cn, yajin_zhou@zju.edu.cn, kuiren@zju.edu.cn
[b] *Department of Computer Science, Virginia Commonwealth University, VA, USA*
E-mails: thaipd@vcu.edu, hszhou@vcu.edu

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 scriptable SNARK, where the prover and verifier(s) can specify the function (or language instance) to be proven via a script. We formalize this notion in 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 SNARK proof systems w.r.t. prover's running time (1000 times faster), verifier's running time, and the proof size. In addition, we also give a lightweight scriptable SNARK protocol for hardware with limited state, e.g., $\Theta(\lambda)$ bits. Finally, we show how the proposed scriptable SNARK 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; on 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.

[1]A preliminary version [60] of this paper was presented at the 26th European Symposium on Research in Computer Security (ESORICS) 2021.
[*]Corresponding author. E-mail: bingsheng@zju.edu.cn.

1.1. Our Design Goals

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 verification time.*

Up to now, those properties have already been achieved by a number of existing NIZK proof systems, such as zk-SNARK [7, 24, 40], zk-STARK [4], 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 needs hours to prove large statement even on a powerful PC (32 cores and 512 GB RAM [4]), let alone portable devices such as smartphones, tablets, and IoT devices. We aim to develop a NIZK system with the property of *(4) 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.

1.2. Our Approach

We propose a new primitive, called *scriptable SNARK*, 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 SNARK system can support multiple language instances, depending on the script language design and the script engine execution environment. Different from existing SNARK systems for universal language, our scriptable SNARK is very easy to use; for a new language instance, the players can easily define the scripts and no further compilation is required.

We study our scriptable SNARK in the UC framework [14, 15]: we define an ideal functionality for scriptable SNARK, and then give two efficient realizations in the trusted hardware model. To the best of our knowledge, there is *no UC-secure SNARK protocol* proposed in the literature. The main reason is that the extractable soundness property of SNARKs in the CRS/RO model require unfalsifiable assumptions [25], such as the knowledge assumption, which is not UC-friendly. Kosba *et al.* [38] has made an attempt on constructing composable NIZK systems, but their protocol is not succinct. To bypass this impossibility result, our protocols utilize a stronger setup assumption, trusted hardware model.

Defining scriptable SNARK. We introduce a new notion called *scriptable SNARK*. Unlike the conventional SNARK, the scriptable SNARK allows the users to specify the relation to be proven via a script. More precisely, the prover only can prove a certain relation in the conventional SNARK, while the prover is allowed to prove any script execution as long as the script is supported.

Formally, 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 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 SNARK 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 SNARK 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 scriptable SNARKs. We then present a generic scriptable SNARK 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., [23] 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 $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ 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 $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ 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 $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$. 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.

Compared with [60]. The construction proposed in the preliminary version [60] requires the trusted hardware to have large enough state that at least linearly depends in the size of the statement $|x|$ and/or the size of the witness $|w|$; moreover, typically, we want to hardware to be programmable. In practice, such powerful trusted hardware is not widely accessible; is it possible to design a scriptable SNARK scheme that can work with trusted hardware with limited state and functionality, such as smart card, USB tokens, etc? In this work, we propose another scheme that can work with trusted hardware with $(\Theta(\lambda))$-size state.

A new construction for lightweight trusted hardware. Note that, in our previous approach, the prover needs to send the entire circuit $\mathcal{C}$ directly to the trusted hardware at once. A natural approach to minimizing the requirement of the hardware state is to disassemble the circuit $\mathcal{C}$ into gates, and we feed the hardware k gates at a time, where k is a small constant, say $k = 1$. Furthermore, we let the potentially malicious prover sends the hash of statement $h_x \leftarrow \mathsf{hash}(\mathsf{Input}_{\mathrm{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{OutPub})$ instead of the statement $\mathsf{Input}_{\mathrm{pub}}, \mathsf{Output}$ to the hardware $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$. Therefore, the input size can be independent to the statement. Meanwhile we re-define the relation $\mathcal{R}' : \mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = \mathsf{Output} \ \wedge \ h_x = \mathsf{hash}(\mathsf{Input}_{\mathrm{pub}}) \ \wedge \ h_o = \mathsf{hash}(\mathsf{Output})$. Without loss of generality, for a relation $\mathcal{R}'$, we assume there exists an efficiently computable algorithm $\mathcal{C}_{\mathcal{R}'}$ such that $\mathcal{C}_{\mathcal{R}'}((h_x, h_o), (\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output})) = 1$ iff $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output} \ \wedge \ h_x = \mathsf{hash}(\mathsf{Input}_{\mathsf{pub}}) \ \wedge \ h_o = \mathsf{hash}(\mathsf{Output})$ and otherwise $\mathcal{C}_{\mathcal{R}'}((h_x, h_o), (\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output})) = 0$. We assume that $\mathcal{C}_{\mathcal{R}'}$ is made of identity gates (for input only) and NAND gates. For such $\mathcal{C}_{\mathcal{R}'}$, it is easy to find a deterministic polynomial-time (DPT) algorithm $\mathsf{Convert}$ which takes $\mathcal{C}$, $\mathsf{Input}_{\mathsf{pub}}$, Output as input and outputs $L := \{L_i\}_{i=1}^{n}$, where L is the description of $\mathcal{C}_{\mathcal{R}'}$, n is the number of the gates and we assume that $k|n$. Then we use hash chains to guarantee the integrity of L. We set $h_L := 0$ at first, and update h_L by $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(i-1)+j}\}_{j=1}^{k}, i)$ for $i \in [\frac{n}{k}]$. To prevent the malicious prover from changing the assignment of wires, we let $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ produce a MAC tag after executing the gate operation, and return the result along with the MAC tag. When the malicious prover sends the input wire value, he/she has to also attach its corresponding MAC tag. When $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ computes the last gate operation and checks the validity, it signs (h_L, h_x, h_o, n) which reveals nothing about $\mathsf{Input}_{\mathsf{priv}}$; therefore, it preserves the zero-knowledge property.

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.* [54], 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 [54]. Most importantly, ours is the first work to investigate *scriptable* SNARK, which is developer-friendly.

1.3. Implementation

We implement our scriptable SNARK 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 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 $\mathsf{Input}_{\mathsf{pub}}$, a private input $\mathsf{Input}_{\mathsf{priv}}$, and a tag tag that can be used to store auxiliary information, such as session id. The Lua engine runs

Table 1

Asymptotic efficiency comparison of different SNARK 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.

Scheme	Setup size	Proof size	Prover's time	Verifier's time	Setup Asm.	Comp. Asm.
Ligero [1]	1	$\sqrt{\lvert C\rvert}$	$\lvert C\rvert \log \lvert C\rvert$	$s \log s + d \log d$	RO	CRHF
Bootle *et al.* [10]	1	$\sqrt{\lvert C\rvert}$	$\lvert C\rvert$	$\lvert C\rvert$	RO	CRHF
Baum *et al.* [2]	$\sqrt{\lvert C\rvert}$	$\sqrt{\lvert C\rvert \log \lvert C\rvert}$	$\lvert C\rvert \log \lvert C\rvert$	$\lvert C\rvert$	CRS	SIS
zk-STARKs [4]	1	$\log^2 \lvert C\rvert$	$\lvert C\rvert\, \mathsf{polylog}(\lvert C\rvert)$	$\mathsf{polylog}(\lvert C\rvert)$	RO	CRHF
Aurora [6]	1	$\log^2 \lvert C\rvert$	$\lvert C\rvert \log \lvert C\rvert$	$\lvert C\rvert$	RO	CRHF
Bulletproof [13]	$\lvert C\rvert$	$\log \lvert C\rvert$	$\lvert C\rvert$	$\lvert C\rvert \log \lvert C\rvert$	CRS + RO	DL
zk-SNARKs [7, 24, 40]	$\lvert C\rvert$	1	$\lvert C\rvert \log \lvert C\rvert$	$\lvert c\rvert$	CRS	KE
This work	1	1	$\lvert C\rvert$	1	HW	Signature

$\mathsf{Output} \leftarrow \mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$ and signs $\langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output} \rangle$. 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 SNARK 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.

We also implement our scriptable SNARK proof system based on trusted hardware with limited state. We simulate the hardware functionality $\mathcal{O}_{\mathsf{HW}}^{\mathsf{Light}}$ on Intel SGX. Most of the instantiations are similar to the one described above, except that: (i) it only computes NAND gates; (ii) it uses MACs. (cf. Sec.6.2)

Performance. The performance of our scriptable SNARK 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.

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.

1.4. Applications

Finally, we discuss applications of our scriptable SNARK. 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. In Sec. 7, we mention a few examples, and showcase how to address these issues via our SNARK. Again, we note that, these issues were not addressed simply due to the missing of fast and SNARK.

Second, we will enable new design paradigm for the interesting "one-to-many" unbalanced computation scenarios. Using our SNARK, 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 SNARK is super succinct, and super fast. In Sec. 7, we demonstrate concrete examples.

2. Preliminaries

2.1. 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. [12, 43], 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) [11]. 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 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.

In practice, the security guarantee of SGX is evolving alone with the discovered side-channel attacks: cache-timing attacks [19], page-fault attacks [59], branch shadowing [41], synchronization bugs [58], foreshadow [12], and SgxPectre [17], etc. Subsequently, some privacy concerns are raised when SGX is involved in the data process. Hereby, we would like to emphasize that unlike most SGX applications, the security of our construction mainly relies on the computational integrity guaranteed by SGX rather than data confidentiality. Namely, the adversary is allowed to learn the enclave's internal state during computation. As far as the root keys are not leaked, the soundness of our construction still holds. This relaxed assumption is modelled as *transparent enclave* in the literature [54].

TrustZone. As one of the most widely deployed security architectures to support TEE, ARM TrustZone separates a processor into two security states, namely the secure world and the normal world. And the resource (e.g., memory, peripherals) belonging to the secure world cannot be accessed by the normal world directly. The processor runs in either the normal world or the secure world at any given time. Switch between the two worlds is triggered by SMC instruction. In this way, ARM TrustZone enables an isolated execution in the secure world. ARM TrustZone can also support attested computation by equipping each TrustZone-enabled device with a device-specific, asymmetric key pair that is signed by the device's vendor. This has been implemented in real-world products (like Samsung Knox). Then by putting the pre-agreed program into the secure world, the system software of the secure world can leverage the private key of the key pair to sign the outcome from the pre-agreed program together with the identity information of the pre-agreed program. Theoretically, anyone can verify the signature of the signed data (outcome and identity information of the pre-agreed program) as long as he or she has the corresponding public key. But in practice, device vendors (like Samsung) tend to maintain an attestation server for signature verification purpose. After the signature passes the verification, the attestation server can generate the corresponding attestation report to indicate that the signed data is indeed produced by a trusted device source, just like what IAS does.

2.2. 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) [8] 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) [27]. 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). Note that, a succinct NIZK proof of knowledge is also called a SNARK. Unfortunately, it is also shown in [25] 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.

2.3. Universal Composibility

Our model is based on the Universal Composibility (UC) framework [14, 15], which lays down a solid foundation for designing and analyzing protocols secure against attacks in an arbitrary *network* execution environment (therefore it is also known as *network aware security model*). Roughly speaking, in the UC framework, protocols are carried out over multiple interconnected machines; to capture attacks, a network adversary $\mathcal{A}$ is introduced, which is allowed to corrupt some machines (i.e., have the full control of all physical parts of some machines); in addition, $\mathcal{A}$ is allowed to partially control the communication tapes of all uncorrupted machines, that is, it sees all the messages sent from and to the uncorrupted machines and controls the sequence in which they are delivered. Then, a protocol Π is a UC-secure implementation of a functionality $\mathcal{F}$, if it satisfies that for every network adversary $\mathcal{A}$ attacking an execution of Π, there is another adversary $\mathcal{S}$—known as the simulator—attacking the ideal process that uses $\mathcal{F}$ (by corrupting the same set of machines), such that, the executions of Π with $\mathcal{A}$ and that of $\mathcal{F}$ with $\mathcal{S}$ makes no difference to any network execution environment $\mathcal{Z}$.
The ideal world execution. In the ideal world, the set of parties $\mathcal{P} := \{P_1, \cdots, P_n\}$ only communicate with an ideal functionality $\mathcal{F}$ and the simulator $\mathcal{S}$ during the execution. In this ideal world, the corrupted parties are controlled by the simulator $\mathcal{S}$. The output of the environment $\mathcal{Z}$ in this execution is denoted by $\mathsf{Exec}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

The real world execution. In the real world, the set of parties $\mathcal{P} := \{P_1, \cdots, P_n\}$ communicate with each other and the adversary $\mathcal{A}$ to run the protocol Π. In this real world, the corrupted parties are controlled by the adversary $\mathcal{A}$. The output of the environment $\mathcal{Z}$ in this execution is denoted by $\mathsf{Exec}_{\Pi,\mathcal{A},\mathcal{Z}}$.

Definition 1. *We say protocol Π UC-secure realizes functionality $\mathcal{F}$ if for all PPT adversaries $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for all PPT environment $\mathcal{Z}$ it holds:*

$$\mathsf{Exec}_{\Pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{Exec}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$$

In order to conceptually modularize the design of the protocols, the notion of "hybrid models" is often introduced in the UC framework. A protocol Π is said to be realized "in the $\mathcal{O}$-hybrid model" if Π invokes the ideal functionality $\mathcal{O}$ as a subroutine.

Definition 2. *We say protocol Π UC-secure realizes functionality $\mathcal{F}$ in the $\mathcal{O}$-hybrid world if for all PPT adversaries $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that for all PPT environment $\mathcal{Z}$ it holds:*

$$\mathsf{Exec}^{\mathcal{O}}_{\Pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{Exec}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$$

The UC model provides strong security guarantees (via polynomial-time security reduction). It also has two appealing features: The property that stand-alone security directly implies security under general concurrent composition (thus protocols only need to be analyzed in a stand-alone fashion), and its support for modular analysis of protocols.

2.4. Cryptographic Tools

We need the following cryptographic tools to build our protocols.

Digital signature. A digital signature DS is a mathematical scheme for verifying the authenticity of digital messages or documents. Formally, the digital signature scheme DS has the following algorithms:

- $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$. It is the key generation algorithm that takes input as the security parameter λ and outputs a public key PK and a signature key SK.
- $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, m)$. It is the signature algorithm that takes input as the signature key SK and a message m. It outputs a signature σ.
- $b \leftarrow \mathsf{DS.Verify}(\mathsf{PK}, m, \sigma)$. It is the verification algorithm that takes input as the public key PK, a message m and a signature σ. It outputs a bit b indicating accpetance $(b = 1)$ or rejection $(b = 0)$.

The security notion required for digital signature is existential unforgeability under chosen message attacks, and we capture it by the following definition.

Definition 3. *We say the digital signature DS is EUF-CMA secure if for any PPT adversary $\mathcal{A}$, the probability of winning the following game $\mathsf{Pr}[\mathsf{Succ}^{\mathsf{DS}}_{\mathcal{A}}]$ is ngeligible*

- *$\mathcal{A}$ interacts with a challenger, denoted as Ch;*
- *Ch runs $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$, and hands the public key PK to $\mathcal{A}$;*
- *For $i \in [n]$: $\mathcal{A}$ sends the message m_i to Ch, Ch runs $\sigma_i \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, m_i)$ and sends σ_i back. Here n is set by $\mathcal{A}$.*

- *$\mathcal{A}$ forges a new message/signature pair (m', σ') and sends it to* Ch*. Then* Ch *checks $m' \notin \{m_i\}_{i=1}^n$ and* $\mathsf{DS.Verify}(\mathsf{PK}, m', \sigma') = 1$*. If both checks pass, $\mathcal{A}$ wins.*

Here we define the advantage of an adversary $\mathcal{A}$ as $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{DS}}(1^\lambda) = \mathsf{Pr}[\mathsf{Succ}_{\mathcal{A}}^{\mathsf{DS}}]$.

Message authentication code. A message authentication code (MAC) is another cryptographic scheme that used to authenticate the origin and nature of a message. It is similar with digital signature, but uses symmetric encryption. A MAC scheme has the following algorithms:

- $\mathsf{K} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$. It is the key generation algorithm that takes input as the security parameter λ and outputs the secret key K.
- $t \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, m)$. It is the signing algorithm that takes input as the secret key K and a message m. It outputs a MAC tag t.
- $b \leftarrow \mathsf{MAC.Verify}(\mathsf{K}, m, t)$. It is the verification algorithm that takes input as the secret key K, a message m and a MAC tag t. It outputs a bit b indicating accpetance $(b = 1)$ or rejection $(b = 0)$.

The security notion required for MAC is the same as digital signature, and we also capture it by the following definition.

Definition 4. *We say the digital signature* MAC *is EUF-CMA secure if for any PPT adversary $\mathcal{A}$, the probability of winning the following game* $\mathsf{Pr}[\mathsf{Succ}_{\mathcal{A}}^{\mathsf{MAC}}]$ *is ngeligible*

- *$\mathcal{A}$ interacts with a challenger, denoted as* Ch*;*
- Ch *runs* $\mathsf{K} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$*;*
- *For $i \in [n]$: $\mathcal{A}$ sends the message m_i to* Ch, Ch *runs* $t_i \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, m_i)$ *and sends t_i back. Here n is set by $\mathcal{A}$.*
- *$\mathcal{A}$ forges a new message/tag pair (m', t') and sends it to* Ch*. Then* Ch *checks $m' \notin \{m_i\}_{i=1}^n$ and* $\mathsf{MAC.Verify}(\mathsf{K}, m', t') = 1$*. If both checks pass, $\mathcal{A}$ wins.*

Here we define the advantage of an adversary $\mathcal{A}$ as $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{MAC}}(1^\lambda) = \mathsf{Pr}[\mathsf{Succ}_{\mathcal{A}}^{\mathsf{MAC}}]$.

Collision-resistant hash function. A hash function hash is collision resistant if it is hard to find two preimages that hash to the same image, that is, two preimages a, b such that $\mathsf{hash}(a) = \mathsf{hash}(b)$. Formally, we capture the collision resistant property by the following definition.

Definition 5. *We say the hash function* hash *is collision resistant if for any PPT adversary $\mathcal{A}$, the probability of winning the following game* $\mathsf{Pr}[\mathsf{Succ}_{\mathcal{A}}^{\mathsf{hash}}]$ *is ngeligible*

- *$\mathcal{A}$ interacts with a challenger, denoted as* Ch*;*
- *$\mathcal{A}$ finds a preimage pair (a, b) and sends it to* Ch*. Then* Ch *checks $a \neq b$ and* $\mathsf{hash}(a) = \mathsf{hash}(b)$*. If both checks pass, $\mathcal{A}$ wins.*

Here we define the advantage of an adversary $\mathcal{A}$ as $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{hash}}(1^\lambda) = \mathsf{Pr}[\mathsf{Succ}_{\mathcal{A}}^{\mathsf{hash}}]$.

3. Security Definition

In this section, we formally define the scriptable SNARK. Our definition is through an ideal functionality $\mathcal{F}_{\mathrm{sSNARK}}^{\mathsf{Q}}$. In addition, we present a setup functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$. We note that the two functionalities will be realized in section 4 and and instantiated in section 6, respectively.

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

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

Proof:

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

Verification:

- Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle)$ from a party $P_j \in \mathcal{P}$:
  - If tuple $\langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle$ is not recorded, send $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle P_j, \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle)$ to $\mathcal{S}$;
  - Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \mathsf{Input_{priv}})$ from $\mathcal{S}$:
    - Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input_{pub}}, \mathsf{Input_{priv}}) = \mathsf{Output}$;
    - Record the tuple $\langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle$;
  - If a tuple $\langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle$ 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 $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$.

Scriptable SNARK ideal functionality. The scriptable SNARK ideal functionality $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ is depicted in Fig. 1. The functionality is parameterized by a predicate Q. The functionality $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ allows the prover to obtain a proof π if $\mathsf{Q}(\mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input_{pub}}, \mathsf{Input_{priv}}) = \mathsf{Output}$, where Q is a predicate. 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 $\mathsf{Input_{priv}}$; therefore, the proof generated by $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ has the conventional zero-knowledge. Since $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ must obtain a private input $\mathsf{Input_{priv}}$ such that $\mathcal{C}(\mathsf{Input_{pub}}, \mathsf{Input_{priv}}) = \mathsf{Output}$ before recording a proof π. Hence, $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ also capture the (knowledge) soundness property. In addition, the scriptable property is reflected by the predicate Q, which restricts the class of functions that $\mathcal{F}^{\mathsf{Q}}_{\text{sSNARK}}$ supports. For instance, Q could be the total execution steps is less than a certain bound.

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

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

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

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 $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}} \rangle)$ from a party $P_i \in \mathcal{P}$ for some ssid, if $(\mathsf{sid}, \cdot, \mathsf{SK})$ is recorded, send $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle P_i, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}} \rangle)$ to the adversary $\mathcal{S}$; Once receiving $(\textsc{Proceed}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$, do:
  - Execute $y \leftarrow \mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$;
  - Assert $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, y) = 1$;
  - Sign $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, y \rangle)$;
  - Return $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle y, \sigma \rangle)$ to the requestor P_i.

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

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 $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$, 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 $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ (cf. Fig. 2, below). As will be shown in Sec. 6 later, it can be instantiated from programmable trusted execution environment (TEE) solutions, e.g., Intel SGX or TrustZone. The $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ functionality is parameterized with a predicate Q and a digital signature scheme, denoted $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$ 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 $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}})$ request to the functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Q}}$, where $\mathcal{C}$ is the polynomial-time algorithm, $\mathsf{Input}_{\mathrm{pub}}$ is the public input, and $\mathsf{Input}_{\mathrm{priv}}$ is the private input. The functionality first computes $\mathcal{C}(\mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}) = y$ and then asserts $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, y) = 1$; it then returns (y, σ), where the signature $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, y \rangle)$. Note that the private input is not signed.

4. Our Scriptable SNARK Construction

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

Common information. Unlike most existing SNARK proof systems, the script $\mathcal{C}$ (or language $\mathcal{L}$ to be proven) is not hardcoded in the prover and verifier executable files. Our scriptable SNARK 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 polynomial 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

Scriptable SNARK protocol $\Pi^{\mathsf{Q}}_{\text{sSNARK}}$

Prove:

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

Verify:

- Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle)$ from the environment $\mathcal{Z}$, $P_j \in \mathcal{P}$ does:
  - *Query $(\textsc{GetPK}, \mathsf{sid})$ to $\mathcal{O}^{\mathsf{Q}}_{HW}$, obtaining $(\textsc{GetPK}, \mathsf{sid}, \mathsf{PK})$;*
  - Parse π as σ;
  - Compute $b \leftarrow \mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}\rangle, \sigma)$;
  - Output $(\textsc{VerifyReturn}, \mathsf{sid}, \mathsf{ssid}, b)$ to the environment $\mathcal{Z}$.

Fig. 3. The scriptable SNARK protocol $\Pi^{\mathsf{Q}}_{\text{sSNARK}}$ in the $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$-hybrid model.

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 SNARK proof systems use different $\mathcal{C}_{\mathcal{R}}$ representation; most popular SNARK 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., [23] 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 $(\mathcal{C}_{\mathcal{R}}, \mathsf{Input_{pub}} := x)$. The potentially malicious prover wants to convince the verifiers that he/she knows a witness $\mathsf{Input_{priv}} := w$ such that $\mathcal{C}_{\mathcal{R}}(\mathsf{Input_{pub}}, \mathsf{Input_{priv}}) = 1$. Since the trusted hardware can guarantee computation integrity even when the host is malicious, we can let $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ to execute the 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 $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ 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 $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$. Similarly, for general computation, the private input $\mathsf{Input_{priv}}$ is not signed; therefore, zero-knowledge property is preserved even if the signature leaks the signed message.

What is the difference between the above SNARK construction and trusted computation in the $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ functionality setting? Recall that SNARK proofs are typically deployed in a one-to-many scenario, so the prover only needs to invoke the $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ 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 $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$.

Construction. Our scriptable SNARK construction utilizes the Q-compliant hardware functionality $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ as defined in Fig. 2.

We aim to achieve constant verification time; light-weight device can perform the verification. In addition, the verifier is only required to query the $\mathcal{O}^{\mathsf{Q}}_{\text{HW}}$ 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 scriptable SNARK proof protocol $\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}$ 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 $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output}) = 1$ and $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output}$; it sends $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$ to $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ and obtains $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathsf{Output}, \sigma \rangle)$ from $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$. 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 $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ to fetch it (marked in grey). In the verification phase, the verifier V accepts the proof if $\mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output} \rangle, \sigma) = 1$.

Security. We show the security of our succinct scriptable SNARK construction via Thm. 1, below.

Theorem 1. *Assume signature scheme* $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *is EUF-CMA secure with adversarial advantage* $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$. *The scriptable NIZK protocol* $\Pi^{\mathsf{Q}}_{sSNARK}$ *described in Fig. 3, UC-realizes the* $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$ *functionality depicted in Fig. 1 in the* $\mathcal{O}^{\mathsf{Q}}_{HW}$*-hybrid world against static malicious corruption with adversarial advantage* $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$.

Proof. To prove the theorem, we construct a simulator $\mathcal{S}$ such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between (i) the real execution $\mathsf{Exec}^{\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}}_{\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{A}, \mathcal{Z}}$ where the parties $\mathcal{P} := \{P_1, \ldots, P_n\}$ run protocol $\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}$ in the $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$-hybrid world and the corrupted parties are controlled by a dummy adversary $\mathcal{A}$ who simply forwards messages from/to $\mathcal{Z}$, and (ii) the ideal execution $\mathsf{Exec}_{\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{S}, \mathcal{Z}}$ where the parties interact with functionality $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$ and corrupted parties are controlled by the simulator $\mathcal{S}$. We consider following cases.

<u>Case 1: The prover P_i is corrupted.</u>

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates honest verifier P_j and the functionality $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

∘ When the simulated $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ receives the incoming message $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}} \rangle)$ from P_i, $\mathcal{S}$ computes $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output}$ and checks if $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output}) = 1$. If it is the case, $\mathcal{S}$ acts as P_i to send $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output} \rangle)$ to $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$. Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle P_i, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output} \rangle)$ from $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$, $\mathcal{S}$ replies $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$, where $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output} \rangle)$ is the signature generated by the simulated $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ functionality.

∘ Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle P_j, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}, \pi \rangle)$ from functionality $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$, the simulator $\mathcal{S}$ checks the internal state (the view) of the simulated $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$, if the command $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}} \rangle)$ has been queried to $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ before, it checks if the generated signature $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output} \rangle)$ is the same as π. If not, aborts. Otherwise, $\mathcal{S}$ returns $\mathsf{Input}_{\mathsf{priv}}$ to $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_1, \mathcal{H}_2$.

Hybrid $\mathcal{H}_1$: It is the real protocol execution $\mathsf{Exec}^{\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}}_{\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{A}, \mathcal{Z}}$.

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, if the proof π is not the same as the simulated signature σ, $\mathcal{S}$ aborts.

Lemma 1. *If* DS *is a EUF-CMA secure digital signature scheme with adversarial advantage* $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$, *then* $\mathcal{H}_4$ *and* $\mathcal{H}_3$ *are indistinguishable with advantage* $\epsilon = \mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$.

Proof. The simulator $\mathcal{S}$ aborts when the prover can generate an accepting proof without querying the functionality $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$. If there exists an adversary $\mathcal{A}$ who can generate an accepting proof without querying $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$, then we can construct an adversary $\mathcal{B}$ who can break the EUF-CMA security game of the underlying digital signature scheme DS. Indeed, since $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ is always trusted, the only way an adversary $\mathcal{A}$ can produce an accepting proof is to forge a signature. During the reduction, $\mathcal{B}$ simulates the $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ as follows. Up on receiving query $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$, $\mathcal{B}$ computes $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output}$ and checks if $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output}) = 1$. It then queries the EUF-CMA game challenger to sign $\langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}\rangle$. After receiving the signature σ from the challenger, $\mathcal{B}$ forwards it to $\mathcal{A}$. When $\mathcal{A}$ outputs $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}^*, \mathsf{Input}^*_{\mathsf{pub}}, \mathsf{Output}^*, \pi^*\rangle)$ such that the verification passes. $\mathcal{B}$ outputs $(m^* := \langle \mathsf{ssid}, \mathcal{C}^*, \mathsf{Input}^*_{\mathsf{pub}}, \mathsf{Output}^*\rangle, \sigma^* := \pi^*)$ as the forged message signature pair. Clearly, $\mathcal{B}$ wins whenever $\mathcal{A}$ wins.

Therefore, the overall advantage is $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$. □

The adversary's view of $\mathcal{H}_2$ is identical to the ideal execution $\mathsf{Exec}_{\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{S}, \mathcal{Z}}$. Therefore, the overall distinguishing advantage is $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$.

<u>Case 2: The verifier P_j is corrupted.</u>

Simulator. Similar to Case 1, the simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates honest prover P_i and the functionality $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

∘ When the simulated $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ receives $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}\rangle)$ from P_i, the simulator $\mathcal{S}$ acts as $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ to generate the corresponding signature $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}\rangle)$ and return $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle \mathsf{Output}, \sigma\rangle)$ to P_i.

Indistinguishability. The indistinguishability is straightforward. The proof π generated by the simulator $\mathcal{S}$ has identical distribution to the proof in the real protocol execution $\mathsf{Exec}^{\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}}_{\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{A}, \mathcal{Z}}$. This is because both proofs are the signatures generated as $\mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}\rangle)$, and the private input is not needed to generate a signature. Moreover, the honest prover first checks the validity of the predicate $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output}) = 1$ and the correctness of the computation output $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output}$ before querying $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$; therefore, the Q-compliance is guaranteed, despite the fact that $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ does not know $\mathsf{Input}_{\mathsf{priv}}$.

<u>Case 3: Both the prover P_i and the verifier P_j are corrupted.</u>

Simulator. Trivial case. There is nothing needs to extract, as the trustees do not have input. The simulator $\mathcal{S}$ just run trustee according to protocol $\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}$.

Indistinguishability. The view of $\mathcal{Z}$ in the ideal execution $\mathsf{Exec}_{\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{S}, \mathcal{Z}}$ has identical distribution to the view of $\mathcal{Z}$ in the real execution $\mathsf{Exec}^{\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}}_{\Pi^{\mathsf{Q}}_{\mathrm{sSNARK}}, \mathcal{A}, \mathcal{Z}}$.

This concludes the proof. □

5. A lightweight SNARK Scheme for Trusted Hardware with Limited State

The hardware functionality $\mathcal{O}_{\mathsf{HW}}^{\mathsf{Q}}$ as described in Fig. 2 requires the trusted hardware to have large enough state that linearly depends in the size of the statement $|x|$ and/or the size of the witness $|w|$; moreover, typically, we want to hardware to be programable. Unfortunately, there are very few products that can fulfill this requirement, which limits the realization choices of our scheme. In this section, we propose another scheme that can work with trusted hardware with $(\Theta(\lambda))$-size state.

Intuition. Our previous approach is to send the entire circuit $\mathcal{C}$ directly to the trusted hardware. A natural approach to minimizing the hardware state requirement is to disassemble the circuit $\mathcal{C}$ into gates, and we feed the hardware k gates at a time, where k is a small constant. However, this would lead to new problems: (i) the trusted hardware with limited state is still not able to load the entire statement x at once; (ii) how to prevent a malicious prover from changing the structure of $\mathcal{C}$ and/or wire assignments.

Our solution is to utilize a collision-resistant hash function hash and the message authentication code (MAC) scheme $\mathsf{MAC} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$. To address the first problem, we send the hash of the statement $h_x \leftarrow \mathsf{hash}(\mathsf{Input}_{\mathsf{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{Output})$ instead of the statement $\mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}$ itself. In this way, we reduce the size of the input. Meanwhile we re-define the relation $\mathcal{R}'$: $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output} \;\wedge\; h_x = \mathsf{hash}(x); \wedge\; h_o = \mathsf{hash}(\mathsf{Output})$. Without loss of generality, for a relation $\mathcal{R}'$, we assume there exists an efficiently computable algorithm $\mathcal{C}_{\mathcal{R}'}$ such that $\mathcal{C}_{\mathcal{R}'}((h_x, h_o), (\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output})) = 1$ if $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output} \;\wedge\; h_x = \mathsf{hash}(x); \wedge\; h_o = \mathsf{hash}(\mathsf{Output})$ and otherwise $\mathcal{C}_{\mathcal{R}'}((h_x, h_o), (\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output})) = 0$. We assume that $\mathcal{C}_{\mathcal{R}'}$ is made of identity gates (for input only) and NAND gates. For such $\mathcal{C}_{\mathcal{R}'}$, we assume it is easy to find a DPT algorithm $\mathsf{Convert}$ which takes $\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}$ as input and outputs $L := \{L_i\}_{i=1}^{n}$, where L is the description of $\mathcal{C}_{\mathcal{R}'}$, n is the number of the gates and we assume that $k|n$. To be specific, the first n_1 items of L are used to describe the identity gates, that is $\{\text{ID } \alpha_i, \gamma_i\}$, where α_i is the input wire of i-th indentity gate while γ_i is the output wire and we assume that $k|n_1$. The rest $n - n_1$ items are used to describe the NAND gates, that is $\{\text{NAND } \alpha_i, \beta_i, \gamma_i\}$, where α_i, β_i are the input wires of i-th NAND gate while γ_i is the output wire. To address the second problem, we use hash chains to guarantee the integrity of L. We set $h_L := 0$ at first, and update h_L by $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(i-1)+j}\}_{j=1}^{k}, i)$ for $i \in [\frac{n}{k}]$. And the prover should send $\langle h_x, h_o, h_L, n\rangle$ at the very beginning.

To prevent the malicious prover from changing the assignment of wires, take NAND gates as an example, the prover has to send the description of the NAND gate $(\alpha_i, \beta_i, \gamma_i)$, the assignment of input wires $x_{\alpha_i}, x_{\beta_i}$ along with their MACs $t_{\alpha_i}, t_{\beta_i}$. The trusted hardware needs to check $\mathsf{MAC.Verify}(\mathsf{K}, \langle \alpha_i, x_{\alpha_i}\rangle, t_{\alpha_i}) = 1$ and $\mathsf{MAC.Verify}(\mathsf{K}, \langle \beta_i, x_{\beta_i}\rangle, t_{\beta_i}) = 1$ first, where $\mathsf{K} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$. After executing NAND operation $x_{\gamma_i} \leftarrow \mathsf{NAND}(x_{\alpha_i}, x_{\beta_i})$, the trusted hardware computes $t_{\gamma_i} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \gamma_i, x_{\gamma_i}\rangle)$, and return $(x_{\gamma_i}, t_{\gamma_i})$.

The lightweight trusted hardware model. Now we provide our new functionality $\mathcal{O}_{\mathsf{HW}}^{\mathsf{Light}}$ in Fig. 4. It is parameterized with a digital signature scheme $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, a MAC scheme $\mathsf{MAC} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, and a hash function hash. It maintains a temporary variable temp and a counter ctr which are both initialized as 0.

$\mathcal{O}_{\mathsf{HW}}^{\mathsf{Light}}$ can be initialized once by sending the $(\textsc{Init}, \mathsf{sid})$ command to it. It then generates $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$, $\mathsf{K} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$ and record $(\mathsf{sid}, \mathsf{PK}, \mathsf{SK}, \mathsf{K})$. After initialization, anyone can query the public key PK using the $\textsc{GetPK}$ command. Anyone, we suppose it is P_i, can then send

$(\textsc{Compute}, \mathsf{sid}, \langle h_x, h_o, h_L, n\rangle)$ request to the functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, where $h_x \leftarrow \mathsf{hash}(\mathsf{Input}_{\mathsf{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{Output})$, h_L is the final output of the hash chain $h_L \leftarrow \mathsf{hash}(h_L, L_i, i)$ for $i \in [n]$, and n is the number of gates. $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ records the tuple $\langle \mathsf{sid}, \mathsf{ssid}, P_i, h_x, h_o, h_L, n\rangle$ for later use. Then P_i can send $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$ and $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$ to $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$. For $\textsc{Id}$ command, $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ simply sets $x_{\gamma_{ku+j}} \leftarrow x_{\alpha_{ku+j}}$, signs $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \mathsf{ssid}, \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$, and returns $\{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^{k}$. For $\textsc{Nand}$ command, $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ checks $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \alpha_{ku+j}, x_{\alpha_{ku+j}}\rangle, t_{\alpha_{ku+j}}) = 1$ and $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \beta_{ku+j}, x_{\beta_{ku+j}}\rangle, t_{\beta_{ku+j}}) = 1$. Then it computes $x_{\gamma_{ku+j}} \leftarrow \mathsf{NAND}(x_{\alpha_{ku+j}}, x_{\beta_{ku+j}})$, signs $x_{\gamma_{ku+j}} \leftarrow \mathsf{NAND}(x_{\alpha_{ku+j}}, x_{\beta_{ku+j}})$, and returns $\{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^{k}$. For both commands, $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ updates $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(i-1)+j}\}_{j=1}^{k}, i)$ and increases $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$. If $\mathsf{temp} = h_L$, $\mathsf{ctr} = \frac{n}{k}$ and $x_{\gamma_n} = 1$ hold, it means $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ has already verified the whole cirucit. $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ then returns σ, where $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle)$. Note that the private input is not signed.

The lightweight scriptable SNARKs construction. Our lightweight scriptable SNARKs construction utilizes the lightweight hardware functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ as defined in Fig. 4.

As depicted in Fig. 5, our lightweight scriptable SNARK protocol $\Pi_{\mathrm{sSNARK}}^{\mathsf{Q,Light}}$ uses a digital signature scheme $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ and hash function hash as its main building block. At the beginning of the protocol, the hardware functionality needs to be initialized, and this step is performed by the prover (marked in grey) if it is not done yet. The prover asserts $\mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output}$ and $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, \mathsf{Output}) = 1$. Note that, the predicate Q here restricts the the class of the relation $\mathcal{R}' : \mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}) = \mathsf{Output} \;\wedge\; h_x = \mathsf{hash}(x); \wedge\; h_o = \mathsf{hash}(\mathsf{Output})$, that is, $\mathcal{R}'$ should be able to be converted to a circuit of polynomial gates. Then the prover computes $h_x \leftarrow \mathsf{hash}(\mathsf{Input}_{\mathsf{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{Output})$ and generates $L \leftarrow \mathsf{Convert}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}})$, where $L := \{L_j\}_{j=1}^{n}$. After that, the prover sets $h_L := 0$, and updates $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(i-1)+j}\}_{j=1}^{k}, i)$ for $i \in [\frac{n}{k}]$. The prover sends $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ to $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, and then sends $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$ for identity gates and $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$ for NAND gates. Finally, the prover obtains $(\textsc{Finish}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ from $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, where 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 $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ to fetch it (marked in grey). In the verification phase, the verifier should computes h_x, h_o and h_L as the prover does, and then accepts the proof if $\mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle, \sigma) = 1$.

Security. We show the security of our lightweight scriptable SNARK construction via Thm. 2, below.

Theorem 2. *Assume the signature scheme* $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *and the MAC scheme* $\mathsf{MAC} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ *are EUF-CMA secure with adversarial advantage* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{DS}}(1^\lambda)$ *and* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{MAC}}(1^\lambda)$ *respectively, and the hash function* hash *is collision resistant with adversarial advantage* $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{hash}}(1^\lambda)$*. The scriptable SNARK protocol* $\Pi_{sSNARK}^{\mathsf{Q,Light}}$ *described in Fig. 5, UC-realizes the* $\mathcal{F}_{\mathrm{sSNARK}}^{\mathsf{Q}}$ *functionality depicted in Fig. 1 in the* $\mathcal{O}_{HW}^{\mathsf{Light}}$*-hybrid world against static malicious corruption with adversarial advantage*

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{MAC}}(1^\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{hash}}(1^\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DS}}(1^\lambda) \ .$$

Proof. To prove the theorem, we construct a simulator $\mathcal{S}$ such that no non-uniform PPT environment $\mathcal{Z}$ can distinguish between (i) the real execution $\mathsf{Exec}_{\Pi_{\mathrm{sSNARK}}^{\mathsf{Q,Light}}, \mathcal{A}, \mathcal{Z}}^{\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}}$ where the parties $\mathcal{P} := \{P_1, \ldots, P_n\}$

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

The functionality interacts with a set of parties $\mathcal{P} := \{P_1, \ldots, P_n\}$ and adversary $\mathcal{S}$. It is parameterized with a digital signature scheme $\mathsf{DS} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, a message authentication code scheme $\mathsf{MAC} := (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, and a hash function hash. It maintains a temporary variable temp and a counter ctr which are both initialized as 0.

- Upon receiving $(\textsc{Init}, \mathsf{sid})$ for the first time from any party $P_i \in \mathcal{P}$:
  - Generate $(\mathsf{PK}, \mathsf{SK}) \leftarrow \mathsf{DS.KeyGen}(1^\lambda)$;
  - Generate $\mathsf{K} \leftarrow \mathsf{MAC.KeyGen}(1^\lambda)$;
  - Record $(\mathsf{sid}, \mathsf{PK}, \mathsf{SK}, \mathsf{K})$;
- 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 $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ from a party $P_i \in \mathcal{P}$, if $(\mathsf{sid}, \cdot, \mathsf{SK}, \mathsf{K})$ is recorded, send $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ to the adversary $\mathcal{S}$; Once receiving $(\textsc{Proceed}, \mathsf{sid}, \mathsf{ssid})$ from $\mathcal{S}$, record the tuple $\langle \mathsf{sid}, \mathsf{ssid}, P_i, h_x, h_o, h_L, n\rangle$, and set $\mathsf{temp} = \mathsf{ctr} = 0$.
- Upon receiving $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^k)$ from a party $P_i \in \mathcal{P}$, if $\langle \mathsf{sid}, \mathsf{ssid}, P_i, h_x, h_o, h_L, n\rangle$ is recorded, do
  - For $j \in [k]$: set $x_{\gamma_{ku+j}} \leftarrow x_{\alpha_{ku+j}}$, and sign $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \mathsf{ssid}, \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$;
  - Increase $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$;
  - Update $\mathsf{temp} \leftarrow \mathsf{hash}(\mathsf{temp}, \{(\alpha_{ku+j}, \gamma_{ku+j})\}_{j=1}^k, \mathsf{ctr})$;
  - Return $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^k)$ to P_i.
- Upon receiving $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^k)$ from a party $P_i \in \mathcal{P}$, if $\langle \mathsf{sid}, \mathsf{ssid}, P_i, h_x, h_o, h_L, n\rangle$ is recorded, do
  - For $j \in [k]$:
    - Assert $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \alpha_{ku+j}, x_{\alpha_{ku+j}}\rangle, t_{\alpha_{ku+j}}) = 1$ and $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \beta_{ku+j}, x_{\beta_{ku+j}}\rangle, t_{\beta_{ku+j}}) = 1$;
    - Compute $x_{\gamma_{ku+j}} \leftarrow \mathsf{NAND}(x_{\alpha_{ku+j}}, x_{\beta_{ku+j}})$;
    - Sign $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \mathsf{ssid}, \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$;
  - Increase $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$;
  - Update $\mathsf{temp} \leftarrow \mathsf{hash}(\mathsf{temp}, \{(\alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j})\}_{j=1}^k, \mathsf{ctr})$;
  - If $\mathsf{ctr} = \frac{n}{k}$, do
    - Assert $x_{\gamma_n} = 1$ and $\mathsf{temp} = h_L$;
    - Sign $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, h_x, h_L, n\rangle)$;
    - Return $(\textsc{Finish}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to P_i;
  - Else, return $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^k)$ to P_i.

Fig. 4. The lightweight trusted hardware functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Light}}$.

run protocol $\Pi_{\text{sSNARK}}^{\mathsf{Q},\mathsf{Light}}$ in the $\mathcal{O}_{\text{HW}}^{\mathsf{Light}}$-hybrid world and the corrupted parties are controlled by a dummy adversary $\mathcal{A}$ who simply forwards messages from/to $\mathcal{Z}$, and (ii) the ideal execution $\mathsf{Exec}_{\mathcal{F}_{\text{sSNARK}}^{\mathsf{Q}}, \mathcal{S}, \mathcal{Z}}$ where the parties interact with functionality $\mathcal{F}_{\text{sSNARK}}^{\mathsf{Q}}$ and corrupted parties are controlled by the simulator $\mathcal{S}$. We consider following cases.

<u>Case 1: The prover P_i is corrupted.</u>

Simulator. The simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates honest verifier P_j and the functionality $\mathcal{O}_{\text{HW}}^{\mathsf{Light}}$. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

○ The simulated $\mathcal{O}_{\text{HW}}^{\mathsf{Light}}$ receives the incoming message $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ from P_i at the beginning. Then the simulated $\mathcal{O}_{\text{HW}}^{\mathsf{Light}}$ will receive $\frac{n}{k}$ messages which are either in form of

A Lightweight Scriptable SNARK Protocol $\Pi^{\mathsf{Q,Light}}_{\mathrm{sSNARK}}$

Prove:

- Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}\rangle)$ from the environment $\mathcal{Z}$, $P_i \in \mathcal{P}$ does:
  - *If the functionality $\mathcal{O}^{\mathsf{Light}}_{HW}$ is not initialized yet, send $(\textsc{Init}, \mathsf{sid})$ to $\mathcal{O}^{\mathsf{Light}}_{HW}$;*
  - Assert $\mathcal{C}(\mathsf{Input_{pub}}, \mathsf{Input_{priv}}) = \mathsf{Output}$ and $\mathsf{Q}(\mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}) = 1$;
  - Compute $h_x \leftarrow \mathsf{hash}(\mathsf{Input_{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{Output})$;
  - Generate $L \leftarrow \mathsf{Convert}(\mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output})$, where $L := \{L_j\}_{j=1}^{n}$;
  - Set $h_L := 0$, and update $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(u-1)+j}\}_{j=1}^{k}, u)$ for $u \in [\frac{n}{k}]$;
  - Send $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ to $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$;
  - For $u = 0, \cdots, \frac{n_1}{k} - 1$, send $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$ to $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ and receive $(\textsc{Id}, \mathsf{sid}, \{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^{k})$;
  - For $u = \frac{n_1}{k}, \cdots, \frac{n}{k} - 1$, send $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$ to $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ and receive $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\rangle\}_{j=1}^{k})$;
  - Obtain $(\textsc{Finish}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ from $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$;
  - Output $(\textsc{ProveReturn}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to the environment $\mathcal{Z}$.

Verify:

- Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}, \pi\rangle)$ from the environment $\mathcal{Z}$, $P_j \in \mathcal{P}$ does:
  - *Query $(\textsc{GetPK}, \mathsf{sid})$ to $\mathcal{O}^{\mathsf{Light}}_{HW}$, obtaining $(\textsc{GetPK}, \mathsf{sid}, \mathsf{PK})$;*
  - Compute $h_x \leftarrow \mathsf{hash}(\mathsf{Input_{pub}}), h_o \leftarrow \mathsf{hash}(\mathsf{Output})$;
  - Generate $L \leftarrow \mathsf{Convert}(\mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output})$, where $L := \{L_i\}_{i=1}^{n}$;
  - Set $h_L := 0$, and update $h_L \leftarrow \mathsf{hash}(h_L, \{L_{k(u-1)+j}\}_{j=1}^{k}, u)$ for $u \in [\frac{n}{k}]$;
  - Parse π as σ;
  - Compute $b \leftarrow \mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle, \sigma)$;
  - Output $(\textsc{VerifyReturn}, \mathsf{sid}, \mathsf{ssid}, b)$ to the environment $\mathcal{Z}$.

Fig. 5. A Lightweight Scriptable SNARK Protocol $\Pi^{\mathsf{Q,Light}}_{\mathrm{sSNARK}}$ in the $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$-hybrid model.

$(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$ or $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$. $\mathcal{S}$ performs the operation according to the protocol of $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$, and finally gets x_{γ_n} and temp. During the simulation of $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$, $\mathcal{S}$ records the view of the simulated $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$. If the prover sends $(\mathsf{ssid}, \alpha, x_\alpha, t_\alpha)$ which is inconsistent with $(\mathsf{ssid}, \alpha, x^*_\alpha, t^*_\alpha)$ yet $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \alpha, x^*_\alpha\rangle, t^*_\alpha) = 1$, $\mathcal{S}$ aborts. In addition, $\mathcal{S}$ can extract $\mathcal{C}$, $\mathsf{Input_{pub}}$, $\mathsf{Input_{priv}}$ and Output from the internal state of the simulated $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$. $\mathcal{S}$ checks if $x_{\gamma_n} = 1$, $\mathsf{temp} = h_L$ and $h_x = \mathsf{hash}(\mathsf{Input_{pub}}), h_o = \mathsf{hash}(\mathsf{Output})$. If it is the case, $\mathcal{S}$ acts as P_i to send $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}\rangle)$ to functionality $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$. Upon receiving $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \langle P_i, \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Output}\rangle)$ from $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$, $\mathcal{S}$ replies $(\textsc{Prove}, \mathsf{sid}, \mathsf{ssid}, \sigma)$ to $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$, where $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle)$ is the signature generated by the simulated $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ functionality. Then $\mathcal{S}$ records the tuple $\langle \mathsf{ssid}, h_x, h_o, h_L, \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}, \sigma\rangle$.

∘ Upon receiving $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle P_j, \mathcal{C}^*, \mathsf{Input}^*_{\mathsf{pub}}, \mathsf{Output}^*, \pi\rangle)$ from functionality $\mathcal{F}^{\mathsf{Light}}_{\mathrm{sSNARK}}$, the simulator $\mathcal{S}$ checks if a tuple $\langle \mathsf{ssid}, h_x, h_o, h_L, \mathcal{C}, \mathsf{Input_{pub}}, \mathsf{Input_{priv}}, \mathsf{Output}, \sigma\rangle$ is recorded. $\mathcal{S}$ computes h^*_L, h^*_x, h^*_o using $\mathcal{C}^*, \mathsf{Input}^*_{\mathsf{pub}}, \mathsf{Output}^*$. If $(\mathcal{C} \neq \mathcal{C}^* \ \wedge \ h_L = h^*_L) \ \vee \ (\mathsf{Input_{pub}} \neq \mathsf{Input}^*_{\mathsf{pub}} \ \wedge \ h_x = h^*_x) \ \vee \ (\mathsf{Output} \neq \mathsf{Output}^* \ \wedge \ h_o = h^*_o)$, $\mathcal{S}$ aborts. If $(h^*_L \neq h_L \ \vee \ h^*_x \neq h_x \ \vee \ h^*_o \neq h_o) \wedge \mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, h^*_x, h^*_o, h^*_L, n\rangle, \pi) = 1$, $\mathcal{S}$ aborts. $\mathcal{S}$ then returns $\mathsf{Input_{priv}}$ to $\mathcal{F}^{\mathsf{Q}}_{\mathrm{sSNARK}}$.

Indistinguishability. The indistinguishability is proven through a series of hybrid worlds $\mathcal{H}_1, \ldots, \mathcal{H}_4$.

Hybrid $\mathcal{H}_1$: It is the real protocol execution $\mathsf{Exec}^{\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}}_{\Pi^{\mathsf{Q,Light}}_{\mathrm{sSNARK}},\mathcal{A},\mathcal{Z}}$.

Hybrid $\mathcal{H}_2$: $\mathcal{H}_2$ is the same as $\mathcal{H}_1$ except that in $\mathcal{H}_2$, during the simulation of $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$, $\mathcal{S}$ records the view of the simulated $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$. If the malicious prover sends $(\mathsf{ssid}, \alpha, x_\alpha, t_\alpha)$ which is inconsistent with $(\mathsf{ssid}, \alpha, x^*_\alpha, t^*_\alpha)$ yet $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{ssid}, \alpha, x^*_\alpha \rangle, t^*_\alpha) = 1$, $\mathcal{S}$ aborts.

Lemma 2. *If* MAC *is a EUF-CMA secure MAC scheme with adversarial advantage* $\mathsf{Adv}^{\mathsf{MAC}}_{\mathcal{A}}(1^\lambda)$*, then* $\mathcal{H}_2$ *and* $\mathcal{H}_1$ *are indistinguishable with advantage* $\epsilon_1 = \mathsf{Adv}^{\mathsf{MAC}}_{\mathcal{A}}(1^\lambda)$.

Proof. The simulator $\mathcal{S}$ aborts when the prover has forged a MAC tag. If there exists an adversary $\mathcal{A}$ who queries the functionality $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ and is able to forge a MAC tag, then we can construct an adversary $\mathcal{B}$ who can break the EUF-CMA security game of the underlying digital signature scheme MAC. During the reduction, $\mathcal{B}$ simulates the $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ as follows. Whenever $\mathcal{B}$ receives Id commands for some ssid and the input wire value x_α, $\mathcal{B}$ sets $x_\gamma \leftarrow x_\alpha$ and sends $\langle \mathsf{ssid}, \gamma, x_\gamma \rangle$ to EUF-CMA game challenger to sign $\langle \mathsf{ssid}, \gamma, x_\gamma \rangle$. After receiving t_γ from the challenger, $\mathcal{B}$ forwards it to $\mathcal{A}$. $\mathcal{B}$ does the similar things when receives Nand commands. Whenever $\mathcal{A}$ outputs $\langle \mathsf{ssid}^*, \gamma^*, x_{\gamma^*}, t_{\gamma^*} \rangle$. $\mathcal{B}$ simply forwards $(m* := \langle \mathsf{ssid}^*, \gamma^*, x_{\gamma^*} \rangle, t^* := t_{\gamma^*})$ to the challenger and wins the game. Clearly, $\mathcal{B}$ wins whenever $\mathcal{A}$ wins.

Therefore, the overall advantage is $\mathsf{Adv}^{\mathsf{MAC}}_{\mathcal{A}}(1^\lambda)$. □

Hybrid $\mathcal{H}_3$: $\mathcal{H}_3$ is the same as $\mathcal{H}_2$ except that in $\mathcal{H}_3$, $\mathcal{S}$ receives $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle P_j, \mathcal{C}^*, \mathsf{Input}^*_{\mathrm{pub}}, \mathsf{Output}^*, \pi \rangle)$ from functionality $\mathcal{F}^{\mathsf{Light}}_{\mathrm{sSNARK}}$, $\mathcal{S}$ founds a recorded tuple $\langle \mathsf{ssid}, \mathcal{C}, \mathsf{Input}_{\mathrm{pub}}, \mathsf{Input}_{\mathrm{priv}}, \mathsf{Output}, \sigma \rangle$ and computes h^*_L, h^*_x, h^*_o using $\mathcal{C}^*, \mathsf{Input}^*_{\mathrm{pub}}, \mathsf{Output}^*$. If $(\mathcal{C} \neq \mathcal{C}^* \ \wedge \ h_L = h^*_L) \ \vee \ (\mathsf{Input}_{\mathrm{pub}} \neq \mathsf{Input}^*_{\mathrm{pub}} \ \wedge \ h_x = h^*_x) \ \vee \ (\mathsf{Output} \neq \mathsf{Output}^* \ \wedge \ h_o = h^*_o)$, $\mathcal{S}$ aborts.

Lemma 3. *If* hash *is a collision resistant hash function with adversarial advantage* $\mathsf{Adv}^{\mathsf{hash}}_{\mathcal{A}}(1^\lambda)$*, then* $\mathcal{H}_3$ *and* $\mathcal{H}_2$ *are indistinguishable with advantage* $\epsilon_2 = \mathsf{Adv}^{\mathsf{hash}}_{\mathcal{A}}(1^\lambda)$.

Proof. The simulator $\mathcal{S}$ aborts when the prover has found a collision for the hash value. If there exists an adversary $\mathcal{A}$ who queries the functionality $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ and finds a collision of h_x, h_o or h_L, then we can construct an adversary $\mathcal{B}$ who can break the collision resistant game of uderlying hash function hash. During the reduction, $\mathcal{B}$ simulates the $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$ as follows. Up on receiving query $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n \rangle)$, $\mathcal{B}$ performs the operation according to the protocol of $\mathcal{O}^{\mathsf{Light}}_{\mathrm{HW}}$, and finally gets x_{γ_n} and temp and the whole collection of $\{L_i\}^n_{i=1}$. First of all, consider the case for h_L. $\mathcal{B}$ forwards h_L to $\mathcal{A}$. $\mathcal{A}$ outputs $L^* := \{L^*_i\}^{n^*}_{i=1}$. $\mathcal{B}$ creates two set $\{h_i\}^n_{i=1}$ and $\{h^*_i\}^{n^*}_{i=1}$, and computes $h_i \leftarrow \mathsf{hash}(h_{i-1}, \{L_{k(i-1)+j}\}^k_{j=1}, i)$ and $h^*_i \leftarrow \mathsf{hash}(h^*_{i-1}, \{L^*_{k(i-1)+j}\}^k_{j=1}, i)$, where $h_0 = h^*_0 = 0$. Whenever $\mathcal{B}$ finds i, j such that $h_i = h^*_j$, $\mathcal{B}$ submits $(\langle h_{i-1}, \{L_{k(i-1)+j}\}^k_{j=1}, i \rangle, \langle h^*_{j-1}, \{L^*_{k(i-1)+j}\}^k_{j=1}, j \rangle)$ to the collision resistant game challenger and wins the game. As for h_x and h_o, similiar approach will be taken. Clearly, $\mathcal{B}$ wins whenever $\mathcal{A}$ wins.

Therefore, the overall advantage is $\mathsf{Adv}^{\mathsf{hash}}_{\mathcal{A}}(1^\lambda)$. □

Hybrid $\mathcal{H}_4$: $\mathcal{H}_4$ is the same as $\mathcal{H}_3$ except that in $\mathcal{H}_4$, If $(h^*_L \neq h_L \ \vee \ h^*_x \neq h_x \ \vee \ h^*_o \neq h_o) \ \wedge$ $\mathsf{DS.Verify}(\mathsf{PK}, \langle \mathsf{ssid}, h^*_x, h^*_o, h^*_L, n \rangle, \pi) = 1$, $\mathcal{S}$ aborts.

Lemma 4. *If* DS *is a EUF-CMA secure digital signature scheme with adversarial advantage* $\mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$*, then* $\mathcal{H}_4$ *and* $\mathcal{H}_3$ *are indistinguishable with advantage* $\epsilon_3 = \mathsf{Adv}^{\mathsf{DS}}_{\mathcal{A}}(1^\lambda)$.

Proof. The simulator $\mathcal{S}$ aborts when the prover can generate an accepting proof without querying the functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$. If there exists an adversary $\mathcal{A}$ who can generate an accepting proof without querying $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, then we can construct an adversary $\mathcal{B}$ who can break the EUF-CMA security game of the underlying digital signature scheme DS with the same advantage. Indeed, since $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ is always trusted, the only way an adversary $\mathcal{A}$ can produce an accepting proof is to forge a signature. During the reduction, $\mathcal{B}$ simulates the $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ as follows. Up on receiving query $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$, $\mathcal{B}$ performs the operation according to the protocol of $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, and finally gets x_{γ_n} and temp. $\mathcal{B}$ checks if $x_{\gamma_n} = 1$ and $\mathsf{temp} = h_L$. It then queries the EUF-CMA game challenger to sign $\langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle$. After receiving the signature σ from the challenger, $\mathcal{B}$ forwards it to $\mathcal{A}$. When $\mathcal{A}$ outputs $(\textsc{Verify}, \mathsf{sid}, \mathsf{ssid}, \langle \mathcal{C}^*, \mathsf{Input}_{\mathrm{pub}}^*, \pi^*\rangle)$ such that the verification passes. $\mathcal{B}$ computes $h_x^* \leftarrow \mathsf{hash}(\mathsf{Input}_{\mathrm{pub}}^*), h_o^* \leftarrow \mathsf{hash}(\mathsf{Output}^*)$ and generates $L^* \leftarrow \mathsf{Convert}(\mathcal{C}^*, \mathsf{Input}_{\mathrm{pub}}^*, \mathsf{Output}^*)$, where $L^* := \{L_i^*\}_{i=1}^{n^*}$. $\mathcal{B}$ then sets $h_L^* := 0$, and updates $h_L^* \leftarrow \mathsf{hash}(h_L^*, \{L^*_{k(i-1)+j}\}_{j=1}^{k}, i)$ for $i \in [\frac{n^*}{k}]$. $\mathcal{B}$ outputs $(m^* := \langle \mathsf{ssid}, h_x^*, h_o^*, h_L^*, n^*\rangle, \sigma^* := \pi^*)$ as the forged message signature pair. Clearly, $\mathcal{B}$ wins whenever $\mathcal{A}$ wins.

Therefore, the overall advantage is $\mathsf{Adv}_{\mathcal{A}}^{\mathsf{DS}}(1^\lambda)$. □

The adversary's view of $\mathcal{H}_4$ is identical to the ideal execution $\mathsf{Exec}_{\mathcal{F}_{\mathrm{sSNARK}}^{\mathsf{Q}}, \mathcal{S}, \mathcal{Z}}$. Therefore, the overall distinguishing advantage is

$$\mathsf{Adv}_{\mathcal{A}}^{\mathsf{MAC}}(1^\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{hash}}(1^\lambda) + \mathsf{Adv}_{\mathcal{A}}^{\mathsf{DS}}(1^\lambda) \ .$$

<u>Case 2: The verifier P_j is corrupted.</u>

Simulator. Similar to Case 1, the simulator $\mathcal{S}$ internally runs $\mathcal{A}$, forwarding messages to/from the environment $\mathcal{Z}$. The simulator $\mathcal{S}$ simulates honest prover P_i and the functionality $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$. In addition, the simulator $\mathcal{S}$ simulates the following interactions with $\mathcal{A}$.

∘ The simulated $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ receives the incoming message $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$ from P_i at the beginning. Then the simulated $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ will receive n messages which are either in form of $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$ or $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$. $\mathcal{S}$ performs the operation according to the protocol of $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$, and finally gets x_{γ_n} and temp. At last, the simulator $\mathcal{S}$ acts as $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ to generate the corresponding signature $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle)$ and return σ to P_i.

Indistinguishability. The indistinguishability is straightforward. The proof π generated by the simulator $\mathcal{S}$ has identical distribution to the proof in the real protocol execution $\mathsf{Exec}^{\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}}_{\Pi_{\mathrm{sSNARK}}^{\mathsf{Q,Light}}, \mathcal{A}, \mathcal{Z}}$. This is because both proofs are the signatures generated as $\mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{ssid}, h_x, h_o, h_L, n\rangle)$, and the private input is not needed to generate a signature.

<u>Case 3: Both the prover P_i and the verifier P_j are corrupted.</u>

Simulator. Trivial case. There is nothing needs to extract, as the trustees do not have input. The simulator $\mathcal{S}$ just run trustee according to protocol $\Pi_{\mathrm{sSNARK}}^{\mathsf{Light}}$.

Indistinguishability. The view of $\mathcal{Z}$ in the ideal execution $\mathsf{Exec}_{\mathcal{F}_{\mathrm{sSNARK}}^{\mathsf{Q}}, \mathcal{S}, \mathcal{Z}}$ has identical distribution to the view of $\mathcal{Z}$ in the real execution $\mathsf{Exec}^{\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}}_{\Pi_{\mathrm{sSNARK}}^{\mathsf{Q,Light}}, \mathcal{A}, \mathcal{Z}}$.

This concludes the proof. □

```
Enclave $\mathcal{SE}$

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

Fig. 6. The script engine enclave $\mathcal{SE}$.

6. Implementation

6.1. $\mathcal{O}_{HW}^{\mathsf{Q}}$ Implementation

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

Challenges. In both platforms, there are a number of challenges need to be resolved. In terms of SGX, as mentioned in Sec. 2.1, the remote attestation of Intel SGX currently requires the verifier to contact the Intel IAS server. On the other hand, in a typical SNARK 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 SNARK proof should be consistent over time, i.e., if a SNARK 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., [54], 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 SNARK proof generated some time ago, how would the verifier know the executable binary (enclave) is faithfully compiled? Therefore, SNARK systems, like [54], 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.

Protocol $\Pi^{\mathsf{Q}}_{\mathrm{SGX}}$

Init

- Upon receiving $(\textsc{Init}, \mathsf{sid})$, the Intel server IS interacts with $\mathsf{HW}_{\mathrm{SGX}}$ invoking the EPID provisioning key procedure (Cf. [35]); At the end of the protocol:
  - ⋆ The Intel server IS stores GPK;
  - ⋆ $\mathsf{HW}_{\mathrm{SGX}}$ 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. 6;
  - ⋆ Sign $\tilde{\sigma} \leftarrow \mathsf{DS.Sign}(\widetilde{\mathsf{SK}}, \mathcal{SE})$;

GetPK

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

Prove

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

Fig. 7. Protocol $\Pi^{\mathsf{Q}}_{\mathrm{SGX}}$ realizing $\mathcal{O}^{\mathsf{Q}}_{\mathrm{HW}}$ via Intel SGX.

SGX-based system overview. In our system, the protocol $\Pi^{\mathsf{Q}}_{\mathrm{SGX}}$ involves three entities: the (trusted) Intel server, denoted as IS, the prover P, and the SGX hardware, denoted as $\mathsf{HW}_{\mathrm{SGX}}$. 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. 6, $\mathcal{SE}$ has one main function called $\mathsf{VerifySign}$[2]. It takes three arguments: (i) a script $\mathcal{C}$, (ii) a public input $\mathsf{Input}_{\mathsf{pub}}$ (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 $\mathsf{Input}_{\mathsf{priv}}$ from the prover; it then executes the script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$ using the script interpreter. Abort if $y = \perp$, which means the execution error happened; that is considered as $\mathsf{Q}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}}, y) = 0$. Otherwise, it sets $h := \mathsf{hash}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, y)$ 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).

Remark. Technically, the private input $\mathsf{Input}_{\mathsf{priv}}$ can be input to the $\mathsf{VerifySign}$ function together with the script $\mathcal{C}$ and the public input $\mathsf{Input}_{\mathsf{pub}}$ as another argument. We choose to load $\mathsf{Input}_{\mathsf{priv}}$ separately

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

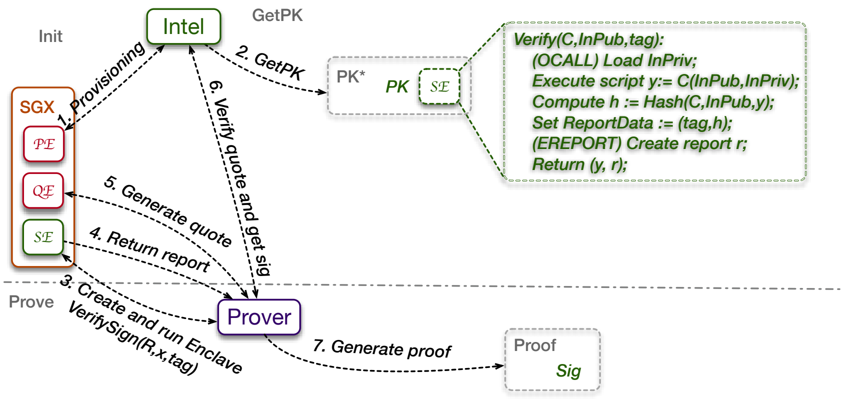


Fig. 8. SGX based trusted hardware instantiation

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. 7. The 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 [35] 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 [35] 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 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. 6 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 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. 7. 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 (y, σ); Fig. 8 summaries the basic flow for the INIT, GETPK, and 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 execution 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 $\langle \mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, y, \mathsf{tag} \rangle$. 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.

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 [50], 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, $\mathsf{VerifySign}$, taking the script $\mathcal{C}$ and the public input $\mathsf{Input}_{\mathsf{pub}}$ as the arguments of the function. In our prototype, the script $\mathcal{C}$ and statement $\mathsf{Input}_{\mathsf{pub}}$ are pre-loaded into the simulated filesystem. After loading $\mathsf{Input}_{\mathsf{priv}}$ from the prover and putting it into the simulated filesystem, the enclave invokes the Lua interpreter to process the script $y \leftarrow \mathcal{C}(\mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$, 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 $h := \mathsf{hash}(\mathcal{C}, \mathsf{Input}_{\mathsf{pub}}, \mathsf{Input}_{\mathsf{priv}})$ and then put (tag, h) in to the $\mathsf{REPORTDATA}$ 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 $\mathsf{ReportBody}$ 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 $\mathsf{isvEnclaveQuoteBody}$, such as $\mathsf{REPORTDATA}$. 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, version, $\mathsf{isvEnclaveQuoteStatus}$. The $\mathsf{isvEnclaveQuoteBody}$ structure is shown in Table 2. The verifier is only interested in the five fields marked in grey background, and

Table 2

QuoteBody Structure

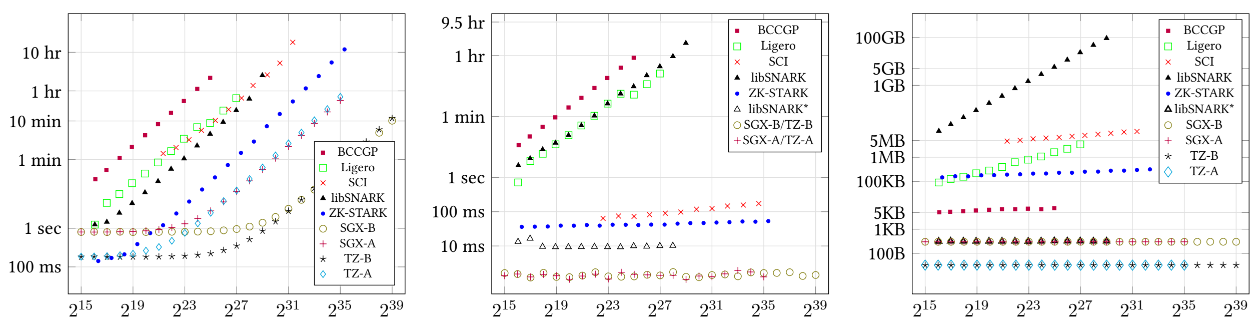
uint16_t	version;
uint16_t	sign_type;
sgx_epid_id_t	epid_group_id;
sgx_isv_svn_t	qe_svn;
sgx_isv_svn_t	pce_svn;
uint32_t	xeid;
sgx_basename_t	basename;
sgx_cpu_svn_t	cpu_svn;
sgx_mise_select_t	misc_select;
uint8_t	reserved1[28];
sgx_attributes_t	attributes;
sgx_measurement_t	mr_enclave;
uint8_t	reserved2[32];
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;

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 $\mathsf{isvEnclaveQuoteStatus} = \mathsf{OK}$; nevertheless, we observe that the attestation verification report whose $\mathsf{isvEnclaveQuoteStatus} = \mathsf{OK}$ 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[3]. 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.

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

[3]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.



(a) Prover Time (b) Verifier Time (c) Proof Size

Fig. 9. Performance comparison of different SNARK proof systems in terms of prover's running time, verifier's running time, and proof size. 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 [4]. 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.

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. 9 shows the performance comparison of different SNARK proof systems w.r.t. prover's running time, verifier's running time, and proof size. Although our SNARK 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 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 [22], SHA256 can be performed at 2.1-3.5GB/s on most platforms.

6.2. $\mathcal{O}_{HW}^{\mathsf{Light}}$ *Implementation*

In this section, we simulate the lightweight trusted hardware functionality $\mathcal{O}_{\mathsf{HW}}^{\mathsf{Light}}$ via Intel SGX. Because most of the instantiations are similiar to Sec. 6.1, except that: (i) it only computes NAND gates; (ii) it also uses MACs, we focus on the differences here.

```
Enclave $\mathcal{E}^{\mathsf{Light}}$

$\mathsf{Init}(h_x, h_o, h_L, n, \mathsf{tag})$ :
  • Set $\mathsf{temp} = \mathsf{ctr} = 0$, and record $\langle h_x, h_o, h_L, n, \mathsf{tag}\rangle$;

$\mathsf{VerifyID}(\{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}, \mathsf{tag})$ :
  • (OCALL) Load $\{x_{\alpha_{ku+j}}\}_{j=1}^{k}$;
  • For $j \in [k]$:
      ∗ Execute $x_{\gamma_{ku+j}} \leftarrow x_{\alpha_{ku+j}}$, and sign the MAC tag $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \mathsf{tag}, \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$;
  • Increase $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$;
  • Update $\mathsf{temp} \leftarrow \mathsf{hash}(\mathsf{temp}, \{(\alpha_{ku+j}, \gamma_{ku+j})\}_{j=1}^{k}, \mathsf{ctr})$;
  • Return $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$;

$\mathsf{VerifyNAND}(\{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}, \mathsf{tag})$ :
  • (OCALL) Load $\{x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\}_{j=1}^{k}$;
  • For $j \in [k]$:
      ∗ Assert $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{tag}, \alpha_{ku+j}, x_{\alpha_{ku+j}}\rangle, t_{\alpha_{ku+j}}) = 1$ and $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{tag}, \beta_{ku+j}, x_{\beta_{ku+j}}\rangle, t_{\beta_{ku+j}}) = 1$;
      ∗ Execute $x_{\gamma_{ku+j}} \leftarrow \mathsf{NAND}(x_{\alpha_{ku+j}}, x_{\beta_{ku+j}})$ , and sign the MAC tag $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \mathsf{tag}, \langle \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$;
  • Increase $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$;
  • Update $\mathsf{temp} \leftarrow \mathsf{hash}(\mathsf{temp}, \{(\alpha_{ku+j}, \gamma_{ku+j})\}_{j=1}^{k}, \mathsf{ctr})$;
  • If $\mathsf{ctr} = \frac{n}{k}$
      ∗ Assert $x_{\gamma_n} = 1$ and $\mathsf{temp} = h_L$;
      ∗ Set $\mathsf{ReportData} = (\mathsf{tag}, h_x, h_o, h_L, n)$;
      ∗ (EREPORT) Create report $r$ for QE to sign;
      ∗ Return $r$;
  • Else, return $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$;
```

Fig. 10. The enclave $\mathcal{E}^{\mathsf{Light}}$.

Enclave. Unlike the script engine enclave depicted in Fig. 6, the enclave $\mathcal{E}^{\mathsf{Light}}$ we create for $\mathcal{O}_{\mathrm{HW}}^{\mathsf{Light}}$ has three main functions: $\mathsf{Init}, \mathsf{VerifyID}, \mathsf{VerifyNAND}$, and it is presented in Fig. 10. The Init function takes (i) the description of the circuit $\mathcal{C}$ and $\mathsf{Input}_{\mathsf{pub}}, \mathsf{Output}$, that is, h_x, h_o, h_L, n; (ii) a tag, tag, that can be used to specify the proof context, such as ssid, etc. The Init function maintains a variable temp and a counter ctr which are both initialized as 0, and records the tuple $\langle h_x, h_o, h_L, n, \mathsf{tag}\rangle$. The $\mathsf{VerifyID}$ function takes (i) the description of the k identity gates, that is $\{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}$ and (ii) a tag tag. The $\mathsf{VerifySign}$ function first loads $\{x_{\alpha_{ku+j}}\}_{j=1}^{k}$ from the prover; it then executes $x_{\gamma_{ku+j}} \leftarrow x_{\alpha_{ku+j}}$ and signs $t_{\gamma_{ku+j}} \leftarrow \mathsf{MAC.Sign}(\mathsf{K}, \langle \mathsf{tag}, \gamma_{ku+j}, x_{\gamma_{ku+j}}\rangle)$. Then it updates $\mathsf{ctr} \leftarrow \mathsf{ctr} + 1$ and $\mathsf{temp} \leftarrow \mathsf{hash}(\mathsf{temp}, \{(\alpha_{ku+j}, \gamma_{ku+j})\}_{j=1}^{k}, \mathsf{ctr})$, and returns $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$. The $\mathsf{VerifyNAND}$ function does the similar things as $\mathsf{VerifyID}$, except that (i) it checks $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{tag}, \alpha_{ku+j}, x_{\alpha_{ku+j}}\rangle, t_{\alpha_{ku+j}}) = 1$ and $\mathsf{MAC.Verify}(\mathsf{K}, \langle \mathsf{tag}, \beta_{ku+j}, x_{\beta_{ku+j}}\rangle, t_{\beta_{ku+j}}) = 1$; (ii) when $\mathsf{ctr} = \frac{n}{k}, x_{\gamma_n} = 1$ and $\mathsf{temp} = h_L$, it sets $\mathsf{ReportData} = (\mathsf{tag}, h_x, h_o, h_L, n)$ and invokes $\mathsf{EREPORT}$ to create a report r for QE to sign.

The Lightweight SNARK System Overview. Here, the protocol $\Pi_{\mathrm{SGX}}^{\mathsf{Light}}$ involves three entities: the (trusted) Intel server, denoted as IS, the prover P, and the SGX hardware, denoted as $\mathsf{HW}_{\mathrm{SGX}}$. We let a

Protocol $\Pi_{\text{SGX}}^{\text{Light}}$

Init

- Upon receiving $(\textsc{Init}, \mathsf{sid})$, the Intel server IS interacts with HW_{SGX} invoking the EPID provisioning key procedure (Cf. [35]); 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 enclave $\mathcal{E}^{\mathsf{Light}}$ as depicted in Fig. 10;
  - Sign $\tilde{\sigma} \leftarrow \mathsf{DS.Sign}(\widetilde{\mathsf{SK}}, \mathcal{E})$;

GetPK

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

Prove

- Upon receiving $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$:
  - The prover P_i creates an enclave instance of $\mathcal{E}$ to HW_{SGX};
  - The prover P_i invokes $\mathsf{Init}(h_x, h_o, h_L, n, \mathsf{tag} := (\mathsf{sid}, \mathsf{ssid}))$;
  - HW_{SGX} sets $\mathsf{temp} = \mathsf{ctr} = 0$ and records $\langle h_x, h_o, h_L, n, \mathsf{tag}\rangle$;
- Upon receiving $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$:
  - The prover P_i creates an enclave instance of $\mathcal{E}$ to HW_{SGX};
  - The prover P_i invokes $\mathsf{VerifyID}(\{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}, \mathsf{tag})$;
    (Supply $\{x_{\alpha_{ku+j}}\}_{j=1}^{k}$ during the execution);
  - HW_{SGX} executes the protocol depicted in Fig. 10 and outputs $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$;
- Upon receiving $(\textsc{Nand}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle, \langle x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\rangle\}_{j=1}^{k})$:
  - The prover P_i creates an enclave instance of $\mathcal{SE}$ to HW_{SGX};
  - The prover P_i invokes $\mathsf{VerifyNAND}(\{\langle \alpha_{ku+j}, \beta_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}, \mathsf{tag})$;
    (Supply $\{x_{\alpha_{ku+j}}, t_{\alpha_{ku+j}}, x_{\beta_{ku+j}}, t_{\beta_{ku+j}}\}_{j=1}^{k}$ during the execution);
  - HW_{SGX} executes the protocol depicted in Fig. 10 and outputs $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$; or a quote $q(\mathsf{tag}, h_x, h_o, h_L, n)$;
  - If HW_{SGX} outputs a quote $q(\mathsf{tag}, h_x, h_o, h_L, n)$:
    - The prover P_i sends the quote $q(\mathsf{tag}, h_x, h_o, h_L, n)$ to the Intel server IS to verify.
    - The Intel server IS checks the validity of the quote; it then signs and returns $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{tag}, h_x, h_o, h_L, n\rangle)$;
    - The prover P_i outputs σ;

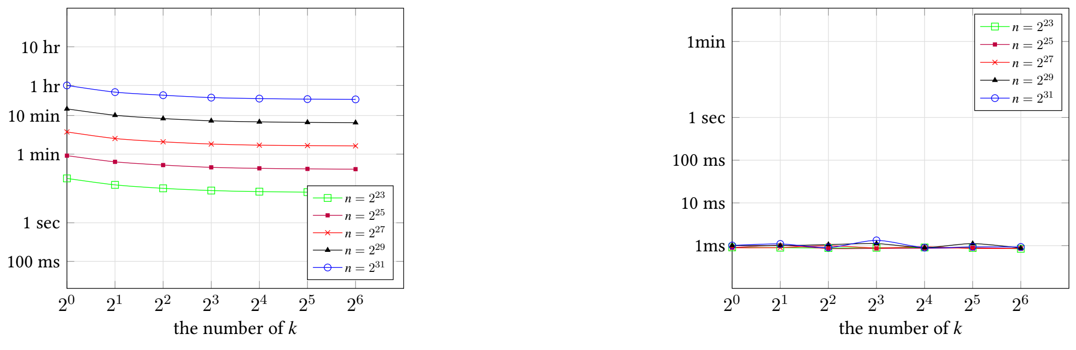
Fig. 11. Protocol $\Pi_{\text{SGX}}^{\text{Light}}$ realizing $\mathcal{O}_{\text{HW}}^{\text{Light}}$ via Intel SGX.

trusted party, i.e., the (trusted) Intel server IS, to produce a enclave $\mathcal{E}^{\mathsf{Light}}$. IS then signs $\mathcal{E}^{\mathsf{Light}}$ so that no one can tamper with its functionality.

The hardware functionality $\mathcal{O}_{\text{HW}}^{\text{Light}}$ is instantiated by the protocol $\Pi_{\text{SGX}}^{\text{Light}}$ shown in Fig. 11. The $\textsc{Init}$ functionality and the $\textsc{GetPK}$ functionality are similiar with protocol $\Pi_{\text{SGX}}^{\mathsf{Q}}$ in Sec. 6.1, thus we will not go into details here. Upon receiving $(\textsc{Compute}, \mathsf{sid}, \mathsf{ssid}, \langle h_x, h_o, h_L, n\rangle)$, the prover P_i creates an enclave instance of $\mathcal{E}^{\mathsf{Light}}$ to HW_{SGX}; it then invokes $\mathsf{Init}(h_x, h_o, h_L, n, \mathsf{tag} := (\mathsf{sid}, \mathsf{ssid}))$. The HW_{SGX} sets $\mathsf{temp} = \mathsf{ctr} = 0$ and records $\langle h_x, h_o, h_L, n, \mathsf{tag}\rangle$. Upon receiving $(\textsc{Id}, \mathsf{sid}, \mathsf{ssid}, \{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle, x_{\alpha_{ku+j}}\}_{j=1}^{k})$, the prover P_i creates an enclave instance of $\mathcal{E}$ to HW_{SGX}; it then invokes $\mathsf{VerifyID}(\{\langle \alpha_{ku+j}, \gamma_{ku+j}\rangle\}_{j=1}^{k}, \mathsf{tag})$ (supply $\{x_{\alpha_{ku+j}}\}_{j=1}^{k}$ during the execution). And the HW_{SGX} executes the protocol depicted in Fig. 10 and outputs $\{x_{\gamma_{ku+j}}, t_{\gamma_{ku+j}}\}_{j=1}^{k}$. The $\textsc{Nand}$ command is similar with the $\textsc{Id}$ command, except when the

HW_{SGX} outputs a report $r(\mathsf{tag}, h_x, h_o, h_L, n)$ for local attestation, the Intel server IS is involved. The prover P_i sends the report $r(\mathsf{tag}, h_x, h_o, h_L, n)$ to the QE of HW_{SGX} to produce a quote $q(\mathsf{tag}, h_x, h_o, h_L, n)$; the prover P_i sends the quote $q(\mathsf{tag}, h_x, h_o, h_L, n)$ to the Intel server IS to verify. The above steps are simplified in Fig. 11. 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 $\sigma \leftarrow \mathsf{DS.Sign}(\mathsf{SK}, \langle \mathsf{tag}, h_x, h_o, h_L, n \rangle)$. The prover P_i outputs σ finally.

Evaluations. Our SGX-based prototype is implemented in C++ using the Intel(R) SGX SDK v2.11 for Linux. we added OpenSSL lib functions for common cryptographic primitives, such as SHA256, ECDSA, etc. We instantiate MAC with HMAC using SHA256 and DS with ECDSA. We expect to achieve two objectives through experiments: (i) try to determine the optimal value of k; (ii) demonstrate the effectiveness of our proposal.



(a) Prover Time (b) Verifier Time

Fig. 12. Performance comparison of different k choices in terms of prover's running time and verifier's running time. The complexity is measured by the number of multiplication gates. Our system is tested on a SGX-equipped processor (i7-8700 @ 3.2GHz and 16GB RAM, single thread).

Fig. 12 shows the performance of our proposal w.r.t. prover's running time and verifier's running time. The complexity is measured by the number of multiplication gates. From the Fig. 12a, we conclude that increasing the value of k at the beginning can significantly improve the prover time, because the initial performance bottleneck lies in the overhead of calling the enclave, and increasing the value of k can reduce the number of calls. As k increases, the curve in Fig. 12a flattens out, which means that the performance bottleneck at this time lies in the computational overhead (e.g. MAC operations) inside the enclave. We also conclude that the performance of our proposal is competitive. When set $n = 2^{25}$, the prover time is less than 25s, and it is much faster than libSNARK, Ligero etc. Note that, no matter which k and n we choose, the proof size is always a small constant (i.e. 297 bytes), thus the verifier time is generally around 1 ms.

7. Blockchain Applications

Resolving verifier's dilemma. The term *verifier's dilemma* in the blockchain context was first proposed in [44]. In this section, we first briefly explain what the problem is and then present a solution using our succinct NIZK proof system.

Table 3

Proving validity of a Bitcoin block (3700 Txs)

Prover time(SGX)	3 s	create enclave	91 ms
		VerifySign	2.2 s
		get QE quote	32 ms
		get IAS report	675 ms
Prover time(TrustZone)	4.348 s	open session	93 ms
		VerifySign	4.241 s
		close session	14 ms
Verifier time(SGX)	1.4 ms		

Verifier's dilemma: In a blockchain system, when a new block is produced, it will be propagated to all the other nodes through the P2P network. In principle, each node needs to independently verify the validity of the block, i.e. in terms of Bitcoin, all the transaction inputs are never spent (e.g., in the UTXO) and the signatures attached to all the transactions are valid. However, in practice, a miner may decide to skip the verification process, for instance, to gain advantages in the proof of work over the other miners – honest miners need to first verify the block, accept it, and then start the proof of work for the next block; whereas, dishonest miners assume that the block is valid, skip the expensive verification, and immediately start to mine the next block.

To resolve the problem, we can let the miner to attach a proof showing the validity of the block. It only takes 1 ms to check the proof; therefore, the disadvantage of honest miners are merely 1 ms, which is negligible compared with the network delay. In our prototype, the statement consists of the root of the Merkle tree commitment (64 levels) of the latest UTXO, denoted as r and the hash of the block, denoted as h. The prover wants to convince the verifiers the followings are true:

- The content of the block that can hash to h;
- For each transaction input, there exist a path of length 64 can be hashed (SHA256) to r;
- The ECDSA signature of each transaction is valid w.r.t. the corresponding public key.

Performance. Table 3 shows the prover's running time to prove the validity of a Bitcoin block with 3700 transactions. For SGX platform, It takes 91 ms to create the enclave, and the VerifySign function running time is 2.2 s. It then takes 32 ms for the QE to sign a quote; it takes approximately 675 ms[4] to contact the IAS and receives the verification report from it. The total time for the prover to generate a proof is about 3 s. Then the verifiers take 1.4 ms to verify the proof. For TrustZone platform, It takes 93 ms to open session with TA in the OPTEE, and invoking VerifySign function takes 4.241 s. It then takes 14 ms to close the session. The total time for the prover to generate a proof is about 4.348 s. Note, in our experiment for TrustZone platform, we omit the interaction time with the remote attestation server because of the lack of the attestation server.

Fast NIPoPoW. Proof-of-Work (PoW) is one of the most popular consensus mechanism to realize an open permissionless blockchain, e.g., Bitcoin and Ethereum. To determine "longest" chain, the nodes need to verify the entire linearly-growing chain of PoWs. Therefore, verify the amount of computation involved in a chain could be an expensive task for a long chain (e.g., the blockchain of Bitcoin consists

[4]The connection time with IAS varies, depending on the region and country. The experiment is tested on a Linode cloud server at Fremont, California, US.

Table 4
Proving chain difficulty for 575000 Bitcoin blocks

Prover time	3.84 s	create enclave	220 ms
		VerifySign	2.89 s
		get QE quote	32 ms
		get IAS report	694 ms
Verifier time	1.5 ms		

of 575000 blocks,) if the nodes only store the genesis block. In practice, checkpoints are used to mitigate this issue.

Non-Interactive Proofs of Proof-of-Work (NIPoPoWs) is a primitive introduced by [37]. It is a short proof that contains the following information,

- the total difficulty of all blocks in a chain,
- if a given block is on that chain.

The verifiers can check the validity of the proof without downloading all the block headers.

NIPoPoWs enables lightweight wallets with simplified payment verification (SPV). The SPV clients can request multiple NIPoPoWs from the full nodes (i.e., the nodes store the whole blockchain). As long as one of those full nodes is honest, the SPV clients can know if a given block is on the longest chain.

NIPoPoWs also can be used to build a cross-chain solution. Because the miners that run a blockchain do not monitor other blockchain networks, this can be done with short proofs. If a blockchain supports smart contracts, e.g., Ethereum, a contract can be written to validate a NIPoPoW to check that something happened on another blockchain and react to it. For instance, a payment made on a blockchain system, that supports NIPoPows, could cause a payment to be released by an Ethereum smart contract.

In the protocol in [37], the miners run an $O\left(|C| \log(|C|)\right)$ (where $|C|$ is the length of the blockchain) algorithm to generate a proof of size $O\left(m \log(|C|)\right)$ where m is a security parameter. After receiving the proof from the miners, the SPV clients can verify the proof with the probability of $1 - \mathsf{negl}(m)$ (where $\mathsf{negl}(m)$ is negligible function of m). The parameter m is a trade-off parameter between security and performance. Increasing m makes the protocol more secure (reduce the probability of false verification), but it also harms the performance (increase the size of the proof and the verification time).

In our SNARK proof system, the miner generate a proof by parsing the blockchain to SGX. The algorithm to generate the proof take $O(|C|)$ complexity. The size of the proof is $O(1)$. The time for the SPV clients to verify the proof is also $O(1)$. Our NIZK proof system is more efficient in terms of proof generation/ verification time and proof size. Furthermore, our verification algorithm always returns the correct value as long as the digital signature scheme is secured. As shown in Table 4, the prover needs about 3.84 seconds to create such a proof (of size 300 bytes) for a chain of 575000 blocks. The verifiers take 1.5 ms to verify the proof.

Privacy preserving smart contract. The smart contract systems over decentralized cryptocurrencies allow making safe transactions between distrustful parties without trusted third parties. However, most of the existing systems lack transaction privacy. All the information of the smart contracts are exposed on the blockchain.

Privacy preserving smart contract is introduced in [36, 39]. Cryptographic primitives, e.g., zero-knowledge proofs, have been used to preserve the privacy of smart contracts. A privacy preserving smart contract consist of two parts

- A private portion which takes in clients' input data (e.g., in two-party coin tossing) as well as currency units (e.g., in an auction). The private portion is executed to determine the payout distribution amongst the clients.
- A public portion (e.g., the smart contract's program) that does not touch private data or money.

After the smart contracts are executed, everyone can verify the execution of the smart contract without knowing any the private portion.

Privacy preserving smart contract can be used for several real-life application, such as insurance, auctions, digital identity and records management. For example, in a unique bid auction smart contract, the clients bid some money to win a prize. The winner is the client with the lowest unique bid. In this case, privacy preserving smart contract is needed so that all the bidding information cannot be revealed.

Nevertheless, the smart contracts in [36, 39] is not effective enough. It takes a few minutes to run the cryptographic computation. Later, there are several papers provide different methods to improve the performance of privacy preserving smart contract. In [18, 61] a trusted execution environment/hardware is combined with a blockchain to address the performance issues. Here, the time to verify an execution of a smart contract can be hundreds of milliseconds.

Our SNARK proof system can also be combined with a blockchain to improve the performance of privacy preserving smart contract. When clients wants to execute a smart contract, they parse the public and private portion to the SGX to generate an execution proof. Then, the miners can verify the proof without knowing the the private portion. We expect the miners can verify the proof within 1.5 ms.

8. Related Work

Universal SNARK. Now we briefly describe several different practical approaches for universal SNARK (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 SNARK can support multiple language instances in a single setup.

There are multiple approaches to scalable SNARK. The first approach is based on homomorphic public-key cryptography, by Ishai et al. [33] and Groth [29]. Then Gennaro et. al [24] introduced an extremely efficient instantiation, based on Quadratic Span Programs, which later been implemented in Pinocchio [49]; see also [5, 7, 20, 40]. 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, [56] 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 SNARK system is more efficient.

The second approach is based on the hardness of the DLP, originally proposed by Groth [30] and then implemented in [9, 13]. 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) [28, 51]. The line of realizations can be found in [62] and [57]. 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. [34] and then implemented in ZKBoo [26], and in Ligero [1]. "MPC in the head" based systems have a non-scalable verifier; in addition, communication complexity is non-scalable.

Not all the existing works can be classified like paragraphs above. Bootle et al. propose a scheme that based on ideal linear commitment (ILC) model where a prover can commit to vectors by sending them to a channel, and a verifier can query the channel on linear combinations of the committed vectors [10]. Baum et al. introduce the first lattice based protocol with sublinear communication costs [2]. A recent proposal called STARK [4], attempts to simultaneously minimize proof size and verifier computation. However, their proof sizes are not small. In [31, 45], 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 [42, 55]. In [5, 7], 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.

NIZK in the UC framework. Groth et al. proposed the first UC-secure NIZK argument for any NP language in the presence of an adaptive adversary [32]. In [32], the simulator is allowed to generate the encryption key/decryption key pair, and encrypts message that relates to the witness. Thus the simulator has the chance of extracting the witness. Since then, a lot of research work has been done to construct the UC-secure NIZK protocol, such as [16]. Kosba et al. has even made an attempt on building a framework for UC-secure NIZK proofs [38]. However, to the best of our knowledge, all of these protocols do not achieve succinctness.

Trusted hardware. Many previous works have proposed using trusted hardware to build cryptographic algorithms and systems, including protection of cryptographic keys [46], functional encryption [23], digital rights management [53], map-reduce jobs [21, 47], machine learning [48], data analysis [52], and protecting unmodified Windows applications [3]. Of course, people have used trusted hardware to build NIZK proof system. More precisely, Tramer *et al.* introduced sealed-glass proof in [54], 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 there are two main difference between their work and ours: interactiveness and scriptability. In particular, 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. Most importantly, ours is the first work to investigate *scriptable* SNARK, which is developer-friendly.

9. Conclusion

In this work, we introduce a new notion called scriptable SNARK proof system. We formally model this notion in the UC framework. We then propose a generic scriptable SNARK 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 scriptable SNARK is better than all the existing succinct SNARK 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). In addition, we also propose a scriptable SNARK solution based on lightweight trusted hardware. Most importantly, our SNARK proof system can be readily deployed and used by any developers without the need of cryptographic background.

References

[1] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2087–2104. ACM Press, October / November 2017.

[2] Carsten Baum, Jonathan Bootle, Andrea Cerulli, Rafaël Del Pino, Jens Groth, and Vadim Lyubashevsky. Sub-linear lattice-based zero-knowledge arguments for arithmetic circuits. In *Annual International Cryptology Conference*, pages 669–699. Springer, 2018.

[3] 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.

[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. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 90–108. Springer, Heidelberg, August 2013.

[6] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. Aurora: Transparent succinct arguments for r1cs. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 103–128. Springer, 2019.

[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, August 2014.

[8] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *20th ACM STOC*, pages 103–112. ACM Press, May 1988.

[9] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 327–357. Springer, Heidelberg, May 2016.

[10] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K Jakobsen. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 336–365. Springer, 2017.

[11] 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, Aug 2010.

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

[13] 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. IEEE Computer Society Press, May 2018.

[14] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/2000/067.

[15] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[16] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive uc nizk. *IACR Cryptol. ePrint Arch.*, 2020:1212, 2020.

[17] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. 2018.

[18] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts.

[19] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[20] 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. Springer, Heidelberg, December 2014.

[21] Tien Tuan Anh Dinh, Prateek Saxena, Ee-Chien Chang, Beng Chin Ooi, and Chunwang Zhang. M2R: Enabling stronger privacy in MapReduce computation. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 447–462. USENIX Association, August 2015.

[22] ECRYPT. ebacs: Ecrypt benchmarking of cryptographic systems. https://bench.cr.yp.to/results-hash.html, 2018. Last accessed: 2019-05-11.

[23] Ben Fisch, Dhinakaran Vinayagamurthy, Dan Boneh, and Sergey Gorbunov. IRON: Functional encryption using intel SGX. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 765–782. ACM Press, October / November 2017.

[24] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[25] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In Lance Fortnow and Salil P. Vadhan, editors, *43rd ACM STOC*, pages 99–108. ACM Press, June 2011.

[26] 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.

[27] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, December 1994.

[28] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: interactive proofs for muggles. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 113–122. ACM Press, May 2008.

[29] Jens Groth. Fully anonymous group signatures without random oracles. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 164–180. Springer, Heidelberg, December 2007.

[30] Jens Groth. Efficient zero-knowledge arguments from two-tiered homomorphic commitments. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 431–448. Springer, Heidelberg, December 2011.

[31] Jens Groth, Markulf Kohlweiss, Mary Maller, Sarah Meiklejohn, and Ian Miers. Updatable and universal common reference strings with applications to zk-SNARKs. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part III*, volume 10993 of *LNCS*, pages 698–728. Springer, Heidelberg, August 2018.

[32] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 339–358. Springer, 2006.

[33] 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.

[34] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In David S. Johnson and Uriel Feige, editors, *39th ACM STOC*, pages 21–30. ACM Press, June 2007.

[35] Simon P Johnson, Vincent R Scarlata, Carlos V Rozas, Ernie Brickell, and Frank McKeen. Intel sgx: Epid provisioning and attestation services. *Intel*, 2016.

[36] Ari Juels, Ahmed E. Kosba, and Elaine Shi. The ring of Gyges: Investigating the future of criminal smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 283–295. ACM Press, October 2016.

[37] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. Non-interactive proofs of proof-of-work. Cryptology ePrint Archive, Report 2017/963, 2017. http://eprint.iacr.org/2017/963.

[38] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, abhi shelat, and Elaine Shi. C∅c∅: A framework for building composable zero-knowledge proofs. Cryptology ePrint Archive, Report 2015/1093, 2015. https://ia.cr/2015/1093.

[39] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy*, pages 839–858. IEEE Computer Society Press, May 2016.

[40] SCIPR Lab. libsnark: a c++ library for zksnark proofs, 2019.

[41] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *USENIX Security*. USENIX Association, 2017.

[42] 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.

[43] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 549–564. USENIX Association, August 2016.

[44] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 706–719. ACM Press, October 2015.

[45] 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.

[46] 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.

[47] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and preventing leakage in MapReduce. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1570–1581. ACM Press, October 2015.

[48] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 619–636. USENIX Association, August 2016.

[49] 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. IEEE Computer Society Press, May 2013.

[50] 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, Piscataway, NJ, USA, 2017. IEEE Press.

[51] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In Daniel Wichs and Yishay Mansour, editors, *48th ACM STOC*, pages 49–62. ACM Press, June 2016.

[52] 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. IEEE Computer Society Press, May 2015.

[53] 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.

[54] 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.

[55] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, pages 196–203, New York, NY, USA, 1976. ACM.

[56] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 1–18. Springer, Heidelberg, March 2008.

[57] 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. IEEE Computer Society Press, May 2018.

[58] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves. In *ESORICS 2016*, pages 440–457, 2016.

[59] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 640–656. IEEE Computer Society, 2015.

[60] Bingsheng Zhang, Yuan Chen, Jiaqi Li, Yajin Zhou, Phuc Thai, Hong-Sheng Zhou, and Kui Ren. Succinct scriptable nizk via trusted hardware. In *European Symposium on Research in Computer Security*, pages 430–451. Springer, 2021.

[61] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. Town crier: An authenticated data feed for smart contracts. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 270–282. ACM Press, October 2016.

[62] 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. IEEE Computer Society Press, May 2017.