IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 9, 2020, accepted November 23, 2020, date of publication November 30, 2020,
date of current version December 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3041308

Practical Verifiable Computation by Using a Hardware-Based Correct Execution Environment

JUNGHEE LEE[1], (Member, IEEE), CHRYSOSTOMOS NICOPOULOS[2], (Member, IEEE), GWEONHO JEONG[3], JIHYE KIM[4], (Member, IEEE), AND HYUNOK OH[3], (Member, IEEE)

[1]School of Cybersecurity, Korea University, Seoul 02841, South Korea
[2]Department of Electrical and Computer Engineering, University of Cyprus, Nicosia 1678, Cyprus
[3]Department of Information System, Hanyang University, Seoul 04763, South Korea
[4]Electronics and Information System Engineering Major, Kookmin University, Seoul 02707, South Korea

Corresponding authors: Jihye Kim (jihyek@kookmin.ac.kr) and Hyunok Oh (hoh@hanyang.ac.kr)

This work was supported by a Korea University Grant and Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Ministry of Science and ICT Korea (2017-0-00661).

ABSTRACT The verifiable computation paradigm has been studied extensively as a means to verifying the result of outsourced computation. In said scheme, the verifier requests computation from the prover and verifies the result by checking the output and proof received from the prover. Although they have great potential for various critical applications, verifiable computations have not been widely used in practice, because of their significant performance overhead. Existing cryptography-based approaches incur significant overhead, because a cryptography-based mathematical frame needs to be constructed, which prevents deviation from the correct computation. The proposed approach is to reduce the overhead by trusting the computing hardware platform where the computation is outsourced. If one trusts the hardware to do the computation, the hardware can take the place of the cryptographic computing frame, thereby guaranteeing correct computation. The key challenge of this approach is to define what exactly the hardware should guarantee for verifiable computation. For this, we introduce the concept of Correct Execution Environment (CEE), which guarantees instruction correctness and state preservation. We prove that these two requirements are satisfactory conditions for a correct output. By employing a CEE, the verifiable computation scheme can be simplified, and its overhead can be reduced drastically. The presented experimental results demonstrate that the execution time is approximately 1.7 million times faster and the verification time over 50 times faster than a state-of-the-art cryptographic approach.

INDEX TERMS Verifiable computation, cryptography, trusted hardware, computer architecture.

I. INTRODUCTION

Verifiable Computation (VC) allows a client (verifier) to delegate the computation of some function F with its outsourced data x to an untrusted third party (prover), while the client can verify the correctness of the result y, i.e., $y := F(x)$. The prover evaluates the functions and returns the result with a proof showing the correctness of the result.

The VC scheme becomes more important as the outsourcing of computations to systems ranging from light computational devices to high-performance cloud-based servers increases [1]. Under VC, it is required that the checking of the correctness of an outsourced function result is much faster than the execution of the function. In addition to cloud services, verifiable computation becomes an essential tool for smart contracts in blockchain applications [2], [3]. In the latter, all transactions that include a smart contract and input/output data become valid after performing the smart contract with the I/O data. Every blockchain node should perform the execution, in order to check the validity of the transaction, which typically incurs high computation overhead. With a plug-in of verifiable computation, the validity check by all blockchain nodes can be accelerated without re-execution of the smart contract. Therefore, the use of a VC scheme improves the efficiency of the blockchain by replacing the execution of the same smart contract on every node with light-weight verification of the execution [2], [3].

To construct VC, cryptographic approaches have been actively researched in the context of zero-knowledge proof systems [4]–[14]. Recently, zk-SNARK (zero-knowledge succinct non-interactive arguments of knowledge) [11]–[14] have been improved significantly, both theoretically and

The associate editor coordinating the review of this manuscript and approving it for publication was SK Hafizul Islam.

practically. They allow a prover to build a proof for any NP statement, while the proof does not leak any information about the witness (or internal values), and the proof size and the verification time are succinct. In addition, it does not require any interaction between the prover (server) and the verifier (client). In practice, anonymous cryptocurrency Zcash [15] adopts the zk-SNARKs to provide privacy by utilizing the zero-knowledge property. Moreover, Zocrates [16] in the Ethereum blockchain has been developed to provide VC by utilizing zk-SNARKs.

Nonetheless, the cryptographic VC approach has a significant performance overhead in the prover (server), even though the verification may be light. Generally, the performance overhead to build a proof ranges from 10,000 to 100,000 times, as compared to a naive computation of the function. Such excessive performance overhead limits its use in practice. In addition, setup (key generation) is required to provide fast verification in zk-SNARKs. Note that the common reference string should be generated in pairing-based zk-SNARKs, in which the proof verification time is constant. Otherwise, the verification time is linear or log-linear to the proof time.

The cryptographic VC approaches generate a cryptography-based mathematical proof to guarantee *the correctness of the output from the given input and function* and *the authenticity of the input, output, and function*. Here, authenticity refers to which input and function the output is generated from and whether they are tampered with or not. If the computing platform where the computation is outsourced guarantees the correctness of the output, the VC scheme only needs to verify the authenticity, which leads to drastic simplification of the scheme. Trusted hardware is a prevalent technique to guarantee security properties by hardware [17]–[24]. However, we will show in Section II that existing trusted hardware is not enough to guarantee security properties required for VC.

In this paper, we introduce the notion of *Correct Execution Environment* (CEE), which guarantees the correctness of the output, and we construct a VC scheme to verify the authenticity of the input, output, and function. A CEE guarantees two requirements: *instruction correctness* and *state preservation*. The former means that each and every instruction works correctly, while the latter means that the architectural state (memory and registers) is preserved in-between the execution of instructions. If the computing platform guarantees these two requirements, we will prove that it guarantees a correct output. The CEE can be implemented either in software or hardware, in various ways, depending on the threat model. We will discuss how to implement a CEE and we will also present our prototypes. If a CEE is employed, the VC scheme can be markedly simplified compared to cryptographic approaches. Through experiments, we will demonstrate that the proposed VC scheme is practical from the perspective of execution and verification times. The execution time of an application with generation of the proof is reduced by up to around 1.7 million times, compared to a state-of-the-art cryptographic approach. In fact, the reduced execution time is only up to 1.1 times longer than the naive execution (without the proof). The verification time is also reduced by up over 50 times, and the reduced verification time is shorter than the naive execution time. This means that the verification is faster than the re-execution of the application. However, for broad adoption of the proposed scheme, the latter needs to overcome the limitation that all the code and data of an application should be located in a designated memory region.

We can summarize the technical contributions of this work, as follows:

- We define primitive requirements to guarantee the correctness of the output for verifiable computation.
- We define and prove the verifiable computation scheme, provided that a verifiable computing platform is used by the prover.
- We propose how to design a verifiable computing platform using hardware components satisfying the requirements.
- We implement verifiable computation both in an AMBER processor (augmented with additional required components), and within Intel's SGX framework [25].
- Through experimental results, we validate that the developed hardware-based verifiable computation is a promising approach.

The rest of this paper is organized as follows: Section II elaborates on the motivation of this paper. Section III defines CEE and discusses how to implement it. Section IV describes the high-level definition of the proposed hardware-based verifiable computation scheme. Section V gives an example of implementation, which is used for the subsequent experiments in Section VI. After discussing relevant related work in Section VII, we conclude this paper in Section VIII.

II. BACKGROUND AND MOTIVATION

The goal of VC is to securely outsource a computation. The client (verifier) requests a computation function (F) with an input (x) to an untrusted server (prover). The prover executes F on x and sends the output (y) along with a proof (π). The verifier should be able to verify $y := F(x)$ by the proof without computing y again.

The proof should be able to prove: (1) the correctness of the output from the given input and function, and (2) the authenticity of the input, output, and function. If the prover is malicious, it may try to deceive the verifier by sending a wrong output. It is also possible for the malicious prover to use a different input or function, instead of the given x and F. The verifier should be able to detect these issues without computing y again.

There have been extensive studies on cryptographic approaches to realize VC. Existing cryptographic approaches prevent deviation from correct computation by forcing the prover to perform the computation within a pre-constructed mathematical frame. Computation in the mathematical frame incurs significant performance overhead, because it requires cryptographic operations. Instead, if the computing platform where the outsourced computation is performed guarantees some security properties, we may not need the expensive mathematical frame to prevent deviation from correct computation. The challenge of this approach, however, is to define what exactly the computing platform should guarantee for the VC.

Is it enough if the computing platform guarantees the integrity of the input and function? The remote attestation technique [19]–[21], [26] can be used to verify such integrity, as well as the integrity of the control flow. When remote attestation is used for embedded systems, the entire memory space can be verified (even though it is still challenging, because of dynamic memory regions), but, for servers, this is not feasible. Typically, only application code and its control flow can be verified; not all the programs running on the same server. However, if not all programs are attested, there is a possibility for a malicious program to distort the output of the attested one. Especially if the malicious program does not change the code or the control flow of the attested program, but only distorts the computation result (also known as data-only attack), it is very challenging to detect this problem by attestation. This is a serious threat to VC, because VC's purpose is to guarantee the correctness of the outsourced computation.

Data-only attacks can be mitigated by a data integrity check [27], [28], or its combination with attestation [23]. However, integrity checks or attestation by themselves do not prove the authenticity of the input, output, and function. The malicious prover may compute y correctly using the correct x and F, but he/she may send a different output, or the output may be tampered with on the way to the verifier. To prevent this, the triple of the input, output, and function should be signed with a signing key to guarantee that the output comes from the given input and function. However, it is not trivial to securely distribute and maintain the key.

For VC, one may also employ Intel's SGX [25], which supports not only attestation, but also memory isolation and hardware-based root-of-trust. SGX encrypts a memory region, which is called Enclave, to prevent other programs – including the privileged system software – from accessing the memory region. When an interrupt occurs, SGX saves the context of the protected function to State Save Area (SSA), which prevents a malicious interrupt handler from distorting the execution of the protected function. Intel employs eFUSE technology to program a random number into each processor, from which the processor derives keys. Therefore, the keys can be securely derived and maintained in the protected memory region. In a prior work, called Sealed-Glass Proof, it is demonstrated that SGX can be used for VC [29].

Nevertheless, we have identified that SGX by itself is still not enough to ensure VC. SGX allows the program running in the Enclave to call an outside function. This is known as OCALL (Outside Call). Since the external function may not be protected by SGX, if the output is dependent on the result of the external function, correct output cannot be guaranteed. Therefore, in order for SGX to be employed for verifiable computation, OCALL should not be allowed.

This observation motivates us to develop a theoretical foundation of trusted hardware for VC. We need a formal definition of requirements that must be guaranteed by the computing platform to guarantee the correctness of the output. There is a prior attempt to formalize the attestation technique [30], but attestation alone does not constitute VC, as discussed above. A Trusted Execution Environment (TEE), such as Intel's SGX, is an isolated execution environment that protects an application by preventing interference from other applications. To develop a TEE, five essential building blocks have been identified [31]. However, they were identified by surveying existing TEEs, not through theoretical, formal deduction. For instance, the aforementioned issue of an OCALL was not addressed in said prior work, because existing TEEs are not specifically intended for VC.

In this paper, we identify two requirements (instruction correctness and state preservation) and prove that they are sufficient to guarantee the correctness of the output. We introduce the concept of *Correct Execution Environment (CEE)*, which guarantees the two requirements. According to our definition, Intel's SGX without OCALL can, indeed, be qualified as a CEE. However, since SGX incurs performance overhead due to additional capabilities that are not essential for VC, we present a new light-weight prototype that only supports VC-essential capabilities.

In general, if the prover runs the outsourced function in a CEE, the correctness of the output is guaranteed. To guarantee the authenticity of the input, output, and function, a VC scheme is constructed. Assuming that the outsourced computation is done within the CEE, the VC scheme is drastically simplified, as compared to cryptographic approaches. The resulting VC scheme is more practical than cryptographic approaches, because it achieves execution times that are close to those of naive execution.

III. CORRECT EXECUTION ENVIRONMENT

In this section, we introduce the concept of a *Correct Execution Environment* (CEE). As previously mentioned, this environment guarantees *instruction correctness* and *state preservation*. We prove that these two requirements are satisfactory conditions to generate correct output. We also discuss how to implement a CEE in practice, since its implementation depends on the threat model.

A. BASICS OF THE VON NEUMANN MODEL

The proposed verifiable computation scheme supports any function implemented as a program that runs on a general-purpose processor that follows the Von Neumann execution model. Unlike previous work [10], [32], our scheme does not dictate a specific type of processor. Instead, the proposed scheme is applicable to any kind of processor, as long as it is compatible with the following description.

The processor architecture supported by our scheme consists of a processor core and a main memory. Instructions, input, output, and temporary data are stored in the memory. The processor core fetches instructions from the memory, executes the instruction, and accesses the memory. The processor core has registers to maintain its status.

Instructions are executed one by one serially. Execution of an instruction is atomic. If an exception occurs, its handler is called only after the current instruction completes its execution. The architectural state is maintained by the memory and registers, which include general-purpose registers and special registers, such as a program counter, stack pointer, link register, and status register. The architectural state does not change after the execution of one instruction, until the next instruction starts.

The description above is the well-known Von Neumann execution model expected by the programmer. However, the hardware does not work in this way. Modern processors often employ pipelining and out-of-order execution. The actual execution of an instruction may be interrupted in the middle of its execution, and the execution order may be different from the program order. Though the actual execution is different, modern processors still follow the Von Neumann execution model, by making the execution of instructions appear to be atomic and in-order. This feat is achieved by employing additional hardware modules, such as a reorder buffer. In addition, exception handling, interrupt service and context switching may result in changes of the state in-between instructions. They are expected to be handled properly so that the state should appear to be preserved.

B. REQUIREMENTS OF CEE

Under the Von Neumann execution model, we can regard the execution of an instruction as a transformation function to the architectural state, and the execution of a program as a series of transformations. The sequence of architectural states of the program execution is called *execution trace* [10]. If we denote the state after execution of the i-th instruction as S_i, the execution trace of a program can be denoted as $(S_0, S_1, S_2, \cdots S_F)$, where S_0 is the initial state and S_F is the final state. We denote the execution of an instruction as a transformation function $S_{out} = I(S_{in})$, where S_{out} is the result of the transformation, I is the transformation function, and S_{in} is the input state. Note that the type of the instruction to be executed (e.g., add, load, store, move, etc.) is also determined by S_{in}, because S_{in} includes the program code stored in the memory and the program counter register.

Let us suppose there is an *ideal machine* that always produces the correct result, and denote its transformation function (execution of an instruction) as I^I, and its execution trace of the given program as $(S_0^I, S_1^I, S_2^I, \cdots S_F^I)$. Then, let us suppose there is a *real machine* whose transformation function as I^R and its execution trace of the same program is $(S_0^R, S_1^R, S_2^R, \cdots S_F^R)$. We will prove that, if the real machine guarantees instruction correctness and state preservation, its execution trace is always the same as that of the ideal machine, thereby qualifying it as a CEE.

Definition 1 (Instruction Correctness): The execution of an instruction is correct if it always produces the same output as the ideal machine for the given input, i.e. $I^R(in) = I^I(in)$.

Definition 2 (State Preservation): The state of a real machine is preserved if the input state of an instruction is always the same as the output state of the previous instruction, i.e., for every i-th instruction, if $S_i^R = I^R(\hat{S}_i^R)$, $\hat{S}_i^R = S_{i-1}^R$.

The definition of state preservation entails memory consistency, because part of the architectural state is maintained by the memory. It is expected that the data written to the memory is always the same as the data fetched later on.

We regard the initial state, S_0, as the primary input of the program to be executed, and the final state, S_F, as the primary output of the program.

Theorem 1: Theorem If a real machine guarantees instruction correctness and state preservation, it guarantees correct output, i.e., the execution trace of the real machine is always the same as that of the ideal machine.

Proof: Let us suppose the same initial architectural state, S_0, which includes the program code, is given to both the ideal machine and the real machine. By instruction correctness, $I^R(S_0) = I^I(S_0)$. Since the type of the first instruction is determined by the initial state, the same instruction is executed both on the ideal machine and the real machine. Both execute the same instruction type with the same input. Therefore, the result is always the same by instruction correctness. In other words, $S_1^R = S_1^I$.

By state preservation, S_1^R is provided as an input to the next instruction on the real machine. Since $S_1^R = S_1^I$, $I^R(S_1^R) = I^I(S_1^I)$, which means $S_2^R = S_2^I$.

By mathematical induction, it is guaranteed that $I^R(S_i^R) = I^I(S_{i-1}^I)$, i.e., $S_i^R = S_i^I$. Therefore, the execution trace of the real machine is always the same as that of the ideal machine, if instruction correctness and state preservation are guaranteed. □

It should be noted that this theorem does not prove the correctness of a program if the program is developed in a wrong way (e.g., it has a bug). The program's output may be different from what is expected by the programmer, but it is still same as the output of the ideal machine. In general, it is the programmer's responsibility to write the program correctly.

C. IMPLEMENTATION OF CEE

The implementation of a CEE varies with the threat model and the processor implementation. Here, we clarify the threat model first, and then discuss how to implement a CEE to satisfy the two requirements.

1) THREAT MODEL

The baseline threat model is *trusted hardware*, i.e., one may trust the hardware and its manufacturer. This threat model is similar to that of Intel's SGX [25].

- Adversaries cannot make changes to the hardware platform.
- Adversaries can access the memory only through the software interface (not by physical or side-channel attacks).
- Adversaries can access the hardware only through the software interface (they cannot read secret information in the hardware, which is kept only in the hardware).
- The manufacturer of the hardware platform is trusted.

Here, the hardware platform includes both the processor core and the memory. We do not assume physical and side-channel attacks to them. In other words, what is not assumed by the threat model above needs to be handled by existing techniques. For example, adversaries may change the content of the memory by a row hammer attack, but it is assumed that such an attack is prevented by existing mechanisms [33]. Embedded processors are often equipped with a JTAG (Joint Test Action Group) interface, which facilitates debugging, but it should be disabled or protected for verifiable computation [34].

This assumption may not be acceptable for every situation, especially if highly capable adversaries are assumed. However, we believe that this assumption would be good enough for most real-world occasions, because hardware attacks demand more resources and capabilities (and sometimes expensive equipment), as compared to software attacks. In fact, existing Trusted Execution Environments (TEE), such as Intel's SGX [25] and ARM's TrustZone [24], are built based on similar assumptions. Since TEE is a widely accepted concept, we believe the trusted hardware threat model is acceptable in most situations.

In addition to the trusted hardware, we may assume that the system software can also be trusted. Table 1 summarizes how to implement a CEE under these two threat models. In the case of instruction correctness, we assume that the trusted manufacturer tries its best to verify the correctness of instructions. State preservation can be realized by trusted hardware or system software. The rest of this subsection discusses how to preserve the architectural state.

TABLE 1. Implementation of correct execution environment.

Property	Trusted H/W	Trusted System S/W
Instruction Correctness	Validated by the trusted manufacturer	
State Preservation (Register)	By hardware backup	By software backup
State Preservation (Memory)	By a hardware module that checks the target address of instructions	By a memory management unit or memory protection unit

2) STATE PRESERVATION (REGISTERS)

If there is only one program running on a computing platform (dedicated platform), there is no possibility of the registers being modified by untrusted software, because none such software runs on the system. However, if an interrupt occurs, the interrupt service routine can modify the registers. Many modern processors employ a separate set of registers for the interrupt mode, but modification of the normal-mode registers is usually allowed to enable exception handling. Thus, even if there is only one program on the platform, any exception handling should not be allowed while the protected program is running.

This constraint, however, may not be acceptable for real-time applications that need to process events in real time. Alternatively, we may backup all registers of the protected program before the interrupt service routine is called, when an interrupt occurs. Register backup can be implemented by hardware or software, depending on the threat model.

This approach can also be applied to multi-process platforms, where multiple programs are running on the same platform. To manage multiple processes, an operating system is also needed. In such a platform, other untrusted programs may modify the registers being used by the protected program. Hence, whenever the protected program is scheduled out in the middle of its execution, all registers should be saved.

Since registers cannot be modified, however, there may be an exception occurring inside the protected program, which cannot be handled by an external exception handler. Recall that it is the programmer's responsibility to develop the program in a correct way. If an exception occurs, the program cannot be recovered and should be terminated.

3) STATE PRESERVATION (MEMORY)

Prevention of illegal modification to the memory is also required. If the processor is equipped with a Memory Management Unit (MMU) or a Memory Protection Unit (MPU), we can use these to control the access to memory regions being used by the protected program. The MMU and MPU are hardware units that grant, or deny, memory access, according to the configuration. Since such units are configured by the system software, if the system software is untrusted we need an additional mechanism to prove the correctness of the configuration. An example of proving correctness is to include the hash of the configuration to the proof that is used to verify the result.

If the processor does not have an MPU, we may add a hardware module that monitors the target address of the load and store instructions and prevents memory access if the target address is in the range of regions used by the protected program and the instruction does not belong to the protected program.

4) EMPLOYING EXISTING TEEs FOR VERIFIABLE COMPUTATION

As discussed in Section II, Intel's SGX can be used to implement a CEE, as long as OCALL is excluded. If OCALL is allowed, and the return value of the external function is used in Enclave, the register that stores the return value is updated by an external function. Since the register is updated outside Enclave, the register appears *not* to be preserved from the perspective of Enclave. If OCALL is not allowed, our definition of CEE confirms that the output generated by SGX is correct, because the architectural state is preserved by the hardware-based memory isolation and the State Save Area (SSA).

Even though it is possible to implement a CEE by using SGX, it incurs significant performance overhead, because the focus of SGX is not solely on VC. Whereas SGX's focus is more on integrity and confidentiality, the focus of a CEE is on correctness. Therefore, SGX incurs significant performance overhead if it is used for verifiable computation, due to the inclusion of capabilities that are not essential for VC. In contrast, the proposed CEE can be implemented in a very light-weight manner. We will demonstrate this through various experiments in Section VI.

D. THE LIMITATION OF A CEE

A CEE does not permit the call of an external function residing outside the program. Thus, the program is required to have all necessary functions inside. Although Intel's SGX allows the call of external functions, calling an external function incurs significant performance overhead, because of expensive context switching [35]. For this reason, Graphene-SGX [35] has been proposed, which makes it easier to develop statically-linked SGX applications. We expect that the same approach can be employed for CEEs.

IV. A VERIFIABLE COMPUTATION SCHEME

By employing a CEE, the *correctness* of the output does not need to be verified. However, the *authenticity* of the input, output, and program must still be verified. In this section, we present a VC scheme for this purpose. Our VC scheme is markedly simpler than cryptographic approaches by assuming that the computation is performed in a CEE.

The proposed VC scheme enables verification of the authenticity of the input, output, and program by a proof. The proof, which is generated by a CEE, is a signature of the concatenation of the input, output, and program. Since only a CEE can generate a valid proof, the proof verifies that: (1) the triple of the input, output, and program is processed *together* in the CEE, (2) the computation is done by a certified CEE, and (3) the integrity of the triple is preserved on the way from/to the verifier.

As a toy example, let us suppose that a verifier wants to outsource the calculation of $1+1$. The proof is a signature on the concatenation of the input, output, and program, i.e., ((1,1), 2, +). The proof verifies that the output (2) is the result of the given particular input and program ((1,1) and +). If the output were produced with a different input or program, the proof would not match. Note that the proof does not verify that the output is the "correct" result. This is guaranteed by the CEE where the computation is outsourced. Since the proof is some kind of digital signature, it proves that the output is generated by a CEE, and, also, that the input, output and program are not tampered with on the way.

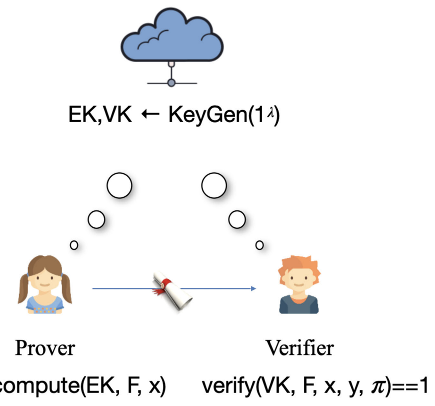

FIGURE 1. Verifiable computation including KeyGen, Compute, and Verify.

A. VERIFIABLE COMPUTATION

In VC, a client delegates the computation of some functions to a server with any input, and the server performs the function on the input and outputs the result and a proof with which the client can verify if the result is correctly computed from the given function and the input. To generate and verify a proof, it is required to generate a key set of evaluation key and verification key, as shown in Figure 1. The server computes the function with an input and generates a proof with the evaluation key. The client finally checks the proof with the verification key.

Cryptographic approaches generate and verify the proof by cryptographic operations, which include multiplication, addition, exponentiation, and modular operations. They also require the verification of the states at every step of computation by calculating the hash values of the states. The execution time of cryptographic approaches is usually very long, because the computation of one arithmetic operation is translated into a large number of cryptographic operations.

Formally, a VC scheme is defined by three algorithms $\mathcal{P} = (KeyGen, Compute, Verify)$, as follows:

- $(EK, VK) \leftarrow KeyGen(1^\lambda)$: The key generation algorithm takes the security parameter λ and generates an evaluation key EK to be used to evaluate the function, and a verification key VK to be used to verify the result.
- $(y, \pi) \leftarrow Compute(EK, F, x)$: The prover takes the evaluation key EK, the function F to be outsourced, and input x; it performs computation of F on x, and generates the result y and its proof π.
- $\{0, 1\} \leftarrow Verify(VK, F, x, y, \pi)$: The verifier takes the verification key VK, the outsourced function F, input x, output y, and the proof π, and verifies that the output is the result of the function on the input, i.e., $y := F(x)$.

The verifier is a client that wishes to learn $y := F(x)$ from the prover, where F is a program that takes an input x and generates an output. F and x can be given by the verifier, or they are publicly available.

The verifiable computation should satisfy the following four properties [36], [37]:

1) COMPLETENESS

It is required that *Verify* $(VK, F, x, y, \pi) = 1$, if $(EK, VK) \leftarrow KeyGen(1^\lambda)$ and $(y, \pi) \leftarrow Compute(EK, F, x)$ for any function F and any input x.

2) SOUNDNESS

For any function F and any probabilistic polynomial-time adversary $\mathcal{A}$, $Pr[F(x) \neq y$ and $1 \leftarrow Verify(VK, F, x, y, \pi)$: $(EK, VK) \leftarrow KeyGen(1^\lambda)$; $(x, y, \pi) \leftarrow \mathcal{A}(EK, VK)] = negl(\lambda)$.

Efficiency. It is required that the time to execute the *Verify* algorithm is asymptotically smaller than the time to evaluate the function F. The proof size needs to be succinct as well; in particular, we require that it is constant, regardless of the function.

3) ZERO-KNOWLEDGE

We also consider a developed setting where the outsourced computation, $F(x, w)$, operates on two inputs: the client's input x and an auxiliary server's input w, which is confidential. A verifiable computation scheme is zero-knowledge if the client cannot learn anything about the server's input w beyond the computation result $F(x, w)$ from a proof. This feature is also useful when a randomized algorithm is outsourced. The random number, or seed, is provided by the server as a confidential input. The proposed VC scheme can be used to prove the correctness of the random algorithm without exposing the random number.

B. DIGITAL SIGNATURE SCHEME

A digital signature scheme is a cryptographic scheme that demonstrates the authenticity of a digital message, or document. The digital signature scheme consists of three algorithms $\mathcal{S} = (Gen, Sig, Ver)$, such that:

- *Gen*: On a security parameter input, it outputs a pair of public/secret keys (PK, SK).
- *Sig*: On inputs of a secret key SK and a message m, it outputs a signature σ.
- *Ver*: On inputs of a public key PK, a message m, and a signature σ, it outputs 1 if the signature σ on the message m is verified with the public key PK; otherwise it outputs 0.

A digital signature scheme is required to satisfy the following two security properties:

1) COMPLETENESS

It is required that for all (PK, SK) output by *Gen*, for all messages M, and for all σ output by $Sig_{SK}(M)$, we have $Ver_{PK}(M, \sigma) = 1$.

2) UNFORGEABILITY

A signature scheme is existentially unforgeable under an adaptive chosen message attack if, for all probabilistic polynomial time attackers $\mathcal{F}$ with access to the signature oracle, the following is negligible in the security parameter: $Pr[Ver_{PK}(M', \sigma) = 1 \wedge M' \notin \mathrm{M} : (PK, SK) \leftarrow Gen; (M', \sigma) \leftarrow \mathcal{F}^{Sig_{SK}(\cdot)}(PK)]$, where M is the set of messages queried by $\mathcal{F}$ to the *Sig* oracle.

C. CONSTRUCTION

The proposed scheme follows the same framework described in the previous subsection. In particular, the scheme is constructed by applying a digital signature scheme to CEE. Given an unforgeable signature scheme $\mathcal{S} = (Gen, Sig, Ver)$, we construct the proposed verifiable computation scheme $\mathcal{P}$, as follows:

- *KeyGen*: On input 1^λ, the algorithm executes $(PK, SK) \leftarrow \mathcal{S}.Gen(1^\lambda)$ and sets $EK := SK$ and $VK := PK$. The evaluation key EK is given to the CEE of the prover. The public key VK is given to the verifier.
- *Compute*: On inputs of the evaluation key EK and the initial architectural state $S_0 = (F, x)$, which includes the program F to be executed and the input data x, the prover's CEE computes the final state $S_F := F(x)$ and generates a proof using EK. The proof is the signature on the initial state S_0 and the final state S_F, i.e., $\pi := \sigma$ where $\sigma \leftarrow \mathcal{S}.SIG_{EK}(S_0||S_F)$. The prover sends the output S_F, and the proof π.
- *Verify*: On inputs (VK, S_0, S_F, π), the verifier verifies the proof by validating the signature with the verification key VK. Only if the signature is validated, i.e., $\mathcal{S}.VER_{VK}(S_0||S_F, \pi)$ outputs 1, the verifier proceeds to verify the result.

Theorem 2: Theorem The above verifiable computation scheme $\mathcal{P}$ is correct, secure, efficient, and zero-knowledge under unforgeability of signatures, instruction correctness, and state preservation of the CEE.

Proof: Completeness holds as follows:

$$\begin{aligned} &Verify(VK, S_0, S_F, \pi) \\ &= S.Ver_{VK}(S_0||S_F, S.Sig_{EK}(S_0||S_F)) \\ &= S.Ver_{PK}(S_0||S_F, S.Sig_{SK}(S_0||S_F)) = 1 \end{aligned}$$

Efficiency holds, since the size of a proof consists of only a single signature element, and the time to verify the proof corresponds to the time to verify one signature, regardless of the function.

The scheme supports the zero-knowledge property. The proof is a signature on the input/output, and it only reveals the validity of the input/output.

Finally, the soundness can be broken either by attacking the inside of the CEE, or by attacking the outside of the CEE. Since the CEE provides instruction correctness and state preservation, signatures by the CEE are generated only on correct execution results. Thus, the former attack is not allowed in the CEE. Said attack may occur on the communication channel between the verifier and the prover (the outside of the CEE). We show that, if there exists an attack to break the soundness of the scheme, then the signature is existentially forgeable. Using the attacker algorithm $\mathcal{A}$ against security, we construct a signature forger algorithm $\mathcal{F}$. Given the public key PK of the signature scheme, $\mathcal{F}$ sets $VK = PK$ and provides VK to $\mathcal{A}$. For the query to the prover on input S_0, $\mathcal{F}$ computes S_F, receives the signature π on $S_0||S_F$ by querying the signature oracle, and responds with S_F, along with π. The query is allowed polynomial number of times. We denote all queries to the prover as $\{(S_0^i, S_F^i, \pi^i)\}_{1\leq i\leq n}$ for n number of queries. Finally, if $\mathcal{A}$ outputs (S_0^*, S_F^*, π^*) s.t. the *Verify* function outputs 1 and $(S_0^*, S_F^*, \pi^*) \notin \{(S_0^i, S_F^i, \pi^i)\}_{1\leq i\leq n}$, then $\mathcal{F}$ outputs $(S_0^*||S_F^*, \pi^*)$, where π^* is a new signature on $(S_0^*||S_F^*)$. By showing the contradiction, the soundness holds. □

Completeness means that the proof must be valid if the computation is performed in an expected way, whereas *soundness* means that the proof must be invalid if the computation is not performed in an expected way. The two requirements of the CEE (instruction correctness and state preservation) guarantee that the prover does not deviate from the expected computation. Thus, the requirements are related to both completeness and soundness. The proof of the proposed VC scheme validates the authenticity, only assuming that the two requirements guarantee that the output is computed in an expected way. The completeness and soundness of the proof hold only if the two requirements are met.

V. PROTOTYPE

As a proof of concept, we present our implementation of the proposed verifiable computation scheme, including implementations of the CEE. The trusted hardware threat model is assumed, and any software – including system software and other programs – is not trusted. Thus, the property of state preservation is supported by hardware mechanisms.

A. OPTIMIZATIONS

To generate the proof, our proposed scheme is required to generate the signature of the architectural state twice, before and after executing the program. The architectural state includes all registers and the entire memory space. Practically, it is not efficient to include all of these.

We can develop the program such that all inputs are given to the memory, and registers are initialized by reading the memory. Thus, the initial state S_0 can be determined only by the initial state of the memory. In this case, we consider the initial state of the memory as an input. In a similar vein, the final output can always be written to the memory when the program finishes. Thus, the final state of the memory is considered as an output. By doing this, we do not have to include registers to generate the proof.

Furthermore, we do not have to include the entire memory space, because not all of it is used by the program. We divide the memory into four regions and require the verifier to provide the memory layout of the program. The memory regions are for inputs (R), outputs (W), temporary data (T), and program code (X). It is required that the input data should be written to R, the program code to X, and the final output to W. The verifier needs to obtain the layout information while the program is being developed. The region for temporary data (T) is used for global variables and dynamic data structures, such as the stack and heap. The program should be developed in such a way that the program code and the input should not be modified, and the final output is written to the designated region (W). The CEE generates the hash on R and X at the beginning of the program, and on W at the end of the program, but not on T, because T is only used for temporary data.

Our prototype supports separate verification. The verifier can verify all of the program code, input, and output, but it may want to verify only a part of them. For example, the verifier may only verify the hash of the program code (excluding the input and output). What it guarantees is that the original program (without any alteration) ran on a known trusted hardware platform. It still gives us useful information if we want to verify the integrity of the program running on a remote device. It is the same principle as the one used in attestation techniques [38]. As another example, we may verify the program code along with the output (excluding the input). It informs us that the output is what the original program has generated on a trusted platform. It can be used to verify the command or data received from a remote device. Thus, the verifier is required to send B, which indicates whether it wants to verify the program code, input and output, or not, individually. If any of hashes are not included in B, it is replaced with all zeros.

In summary, S_0 in our prototype consists of the following fields:

- X: Program code.
- R: Input data.
- L: Memory layout.
- B: Flags that indicate inclusion of X, R, and W.

For efficient computation of hashes, we compute the hashes of the four fields individually, and concatenate them when they are signed. In the case of the output, S_F is the same as W.

B. PROTOCOL

The protocol is defined as follows (also illustrated in Figure 2):

- **(S1)** The manufacturer programs a random number into the CEE when the CEE is manufactured.
- **(S2)** The CEE derives a pair of public and private keys and publishes the public key.
- **(S3)** The verifier sends the initial state to the prover, which includes program code, input data, memory layout, and flags.
- **(S4)** The prover writes the program code and the input data to the designated memory regions.
- **(S5)** The prover initializes the CEE by setting the memory layout information.
- **(S6)** When the program starts, the CEE generates the hashes on R and X. It also generates the hash on the memory layout information.

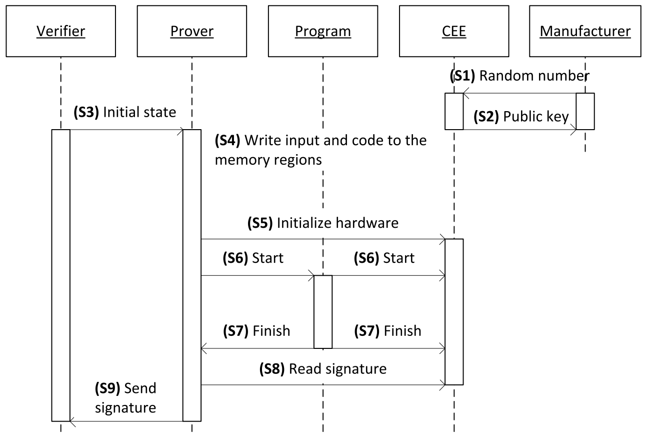

FIGURE 2. **The protocol of the proposed verifiable computation scheme when a CEE is employed.**

- **(S7)** When the program finishes, the CEE generates the hash on W and a signature on the concatenation of all the hashes.
- **(S8)** The prover reads the signature and sends it to the verifier, along with the output and the public key of the CEE.
- **(S9)** The verifier verifies the signature by the public key and the output.

In our prototype, the CEE is implemented in hardware and it generates the signature. As Intel does for SGX, we also assume that the manufacturer programs a random seed so that the CEE can derive a pair of public and private keys. It is assumed that the trusted manufacturer honestly publishes the public key received from the CEE. Unless physical attacks are committed, the private key is securely kept only in the hardware of the CEE. The private key cannot be read through the software interface.

Our threat model assumes the prover is not trusted. The prover may try to deceive the verifier by modifying the program or input data, writing them to a wrong memory region, giving a wrong memory layout to the hardware, or forge the output. However, it cannot forge the proof, which is signed by the hardware, because the private key is not accessible. Therefore, all of these trials result in a discrepancy in the proof, which can be detected by the *Verify* function.

C. CEE PROTOTYPE

We prototype the CEE under the assumption of the trusted hardware (but not system software), as described in Section III-C. We implement it on an open-source AMBER processor which is written in Verilog [39]. The instruction correctness property is validated by extensive simulations. The state preservation property is realized by the techniques presented in the second column of Table 1, which refer to the trusted hardware threat model. The register file is modified to preserve registers. The execution stage is modified to prevent illegal modification to memory regions.

For verifiable computation, we add a hardware accelerator to generate the signature. The cache controller is modified to compute the hashes of X, R, and W, and a dedicated encryption module is added to generate the signature.

Finally, we add a memory-mapped peripheral connected to the system bus, which interacts with the software and provides necessary information to the hardware logic, which is added to implement the proposed scheme.

1) STATE PRESERVATION

If the program counter exits from region X while the protection mode is active, all registers should be saved. In our implementation, we employ extra hardware registers. If the registers are saved by software, they may be saved in the memory. However, since this process should be performed by the hardware, the latter cannot handle the situation if there is not enough space in the memory. If this situation occurs, the hardware platform may not be able to guarantee correct output.

We employ a backup register for every register in the processor. When the program counter exits from X, all registers are saved to their corresponding backup registers, and they are restored from the backup registers when the program counter comes back. Even though this approach incurs hardware cost, it does not incur any performance overhead, because the registers are saved and restored in one clock cycle.

While the protection mode is active, the target address of load and store instructions, and their program counter should be checked. Depending on the addressing mode, the target address may become available at the decode stage, or the execution stage. Since the execution stage is after the decode stage, the target address is certainly available in the execution stage. Thus, we implement the address checking logic in the execution stage.

The logic checks the conditions of the memory target address and the program counter. The logic prevents the memory from being modified by other programs. This can be achieved by monitoring the store instructions. Here, a ‘‘store’’ instruction refers to all instructions that write data to the memory. If the target address of the store instruction is in W or T, the location (program counter) of that instruction must be in X.

Note that R and X cannot be written by any program. The protected program itself is required not to modify its program code and the input. This is required for efficient generation of hashes. This requirement is also checked by monitoring the store instructions. If the location of the store instruction is in X, its target address must be in W or T. However, it is still the responsibility of the program developer to ensure the final output is written to W. If the final output is written only to T, but not to W, its hash cannot be generated.

2) SIGNATURE GENERATION

While the hash of region W needs to be computed after the program completes, the hashes of R and X can be computed anytime, because we do not allow modifications to R and X. If self-modifying code is allowed, the hash of X needs to be computed before the program starts. In this case, we cannot exploit the cache. If the hash is computed during the execution of the program, the data in the cache can be used for hash computation, which reduces the performance overhead. Thus, we implement hash computation by modifying the cache controller.

If the processor needs to access the memory while the hash computation logic is accessing the memory, the processor should wait, which causes performance degradation. Consequently, the hash computation logic needs to be designed so as to minimize interference with the processor.

In the case of AMBER's cache controller, it does not support the handling of multiple outstanding cache misses. When a cache miss occurs, the processor is stalled and does not access the cache. When a cache hit occurs, the cache controller does not access the memory. We exploit this to minimize the interference.

The hash computation logic searches for data in the cache only while a cache miss occurs. It reads data from the memory only while a cache hit occurs. Since memory access cannot be preempted, this implementation still incurs overhead if a cache miss occurs before the hash computation logic finishes its memory access. However, this approach minimizes the interference by exploiting the idle time of the cache controller.

The computed hashes are sent to another hardware logic module that generates the signature after the program finishes.

3) MEMORY-MAPPED PERIPHERAL

The memory-mapped peripheral is an interface with the prover software. It provides a set of registers that act as the desired interface. Table 2 summarizes these registers.

TABLE 2. Registers of the memory-mapped peripheral.

Name	Read/Write	Description
`R_START`	Read/Write	The start address of region R
`R_END`	Read/Write	The end address of region R
`X_START`	Read/Write	The start address of region X
`X_END`	Read/Write	The end address of region X
`W_START`	Read/Write	The start address of region W
`W_END`	Read/Write	The end address of region W
`T_START`	Read/Write	The start address of region T
`T_END`	Read/Write	The end address of region T
`START`	Write Only	The prover writes 1 when it is ready to start computation
`DONE`	Write Only	The program writes 1 when it finishes computation
`READY`	Read Only	The hardware writes 1 when it finishes signature generation
`SIG`	Read Only	The generated signature
`KEY_P`	Read Only	The public key of the platform

The first eight registers are for setting the memory regions. The prover sets them up after receiving the pertinent information from the verifier. The prover also needs to store the program code and input to the designated memory regions. After it finishes storing, it writes 1 to `START` and then jumps to the entry point of the program in region X. The hardware activates the protection mode when the program counter enters region X, while `START` is 1. While the program is running, the hardware computes the hashes of X and R. The hardware platform guarantees that the contents in X and R will not change until the program completes, by checking the memory target address and program counter, as previously discussed. When the program is done, it writes 1 to `DONE`, which informs the hardware that the output is available. The hardware computes the hash of W and generates the signature. After generating the signature, the hardware platform writes the signature to `SIG` and 1 to `READY` to inform the prover that the signature is ready to read. The prover reads the signature and sends it to the verifier along with the output. When the prover reads `SIG`, the hardware platform terminates the protection mode and resets its state to get ready for the next computation. When the verifiable computation is terminated, the memory regions (R, X, W, and T) are freed.

4) EXTENSION

Our CEE prototype employs dedicated hardware accelerators for crytographic primitives (hash and digital signature). If the VC scheme is extended to support other cryptographic algorithms, we may employ flexible hardware accelerators [40], [41]. In addition, the CEE can protect multiple applications simultaneously, if the CEE is equipped with multiple instances of the memory-mapped peripheral.

D. CEE PROTOTYPE WITH INTEL's SGX

We also prototype the VC scheme using Intel's SGX under the same trusted hardware assumption. Even though SGX encrypts the memory, which may appear to be stronger than the CEE requirements, it does not affect the threat model. If adversaries can make changes to the hardware, SGX cannot be trusted. If they access the memory by other means than the software interface, the adversaries cannot read the contents, but they can still *tamper* with the contents, which could result in incorrect outputs. If the adversaries can access the secret information in the hardware through physical attacks, SGX cannot be trusted. If the manufacturer is not trusted, we cannot authenticate SGX-enabled processors.

Under this threat model, as long as OCALL is not allowed, SGX is qualified as a CEE and guarantees the correctness of the output, but it does not guarantee the authenticity of the input, output, and program. Thus, we implement our VC protocol in Enclave. The hash computation and signature generation are implemented as software running in Enclave.

The SGX-based prototype is intended to demonstrate that our definition of CEE is general enough to be implemented in various ways, and to compare the performance overhead against our light-weight prototype.

VI. EXPERIMENTAL EVALUATION

Through various experiments, we demonstrate that the performance overhead of the proposed scheme is an order-of-magnitude lower than prior work. We also analyze the hardware cost to implement the hardware-based CEE.

The performance is measured by running benchmarks on the AMBER processor. The number of clock cycles is measured by simulating the processor. Since the processor is written in Verilog, we use the Xilinx ISE Verilog simulator.

We use 7 benchmarks from the MiBench suite [42]. The MiBench suite is mainly targeted at the evaluation of embedded systems, and it consists of various algorithms, including cryptography (BlowFish, SHA), media (ADPCM), encoding (CRC32), sort (QuickSort), bit counting (BitCount), and string matching (StringSearch). The benchmarks are compiled using the GCC ARM cross compiler, because the AMBER processor is compatible with the ARM ISA. The MiBench benchmark characteristics are summarized in Table 3. Even though the proposed scheme does not impose any limitations on the program size and input size (other than the physical memory size of the platform), we limit the size of the benchmarks, in order to compare the results with previous works (to be presented later on). Said previous works impose limitations on the program and input sizes.

TABLE 3. Characteristics of MiBench [42] benchmarks. 'Executed' refers to the number of executed instructions.

Benchmark	Executed	Program size	Input size
ADPCM	121,672	5,116 B	3,072 B
BitCount	115,895	4,828 B	292 B
BlowFish	372,860	4,820 B	4,258 B
CRC32	137,460	4,576 B	1,136 B
QuickSort	126,712	4,320 B	528 B
SHA	239,833	5,968 B	672 B
StringSearch	167,549	4,444 B	2,852 B

To analyze the hardware cost, we measure the hardware area by using the Synopsys Design Compiler. We compare the area of the modified processor against the original one.

The proposed scheme is orthogonal to the hashing and signing algorithms. Thus, we estimate their impact on performance and hardware cost from the latest literature [43]. SHA-2 (Secure Hash Algorithm) [44] and ECDSA (Elliptic Curve Digital Signature Algorithm) [45] are used as hashing and signing algorithms, respectively, in our experiments.

A. PROVER OVERHEAD

The prover needs to generate the proof while running the program. In this subsection, the overhead of proof generation is analyzed. The overhead is measured by comparing the execution time of a benchmark with and without proof generation.

In the proposed scheme, the proof is generated by the hardware platform. The overhead occurs if the hash computation logic interferes with the processor when it accesses the memory. Thus, the overhead is positively correlated with the memory access time. In other words, if the memory access time increases, the overhead is also likely to increase. From the perspective of the processor, if the cache size is smaller,

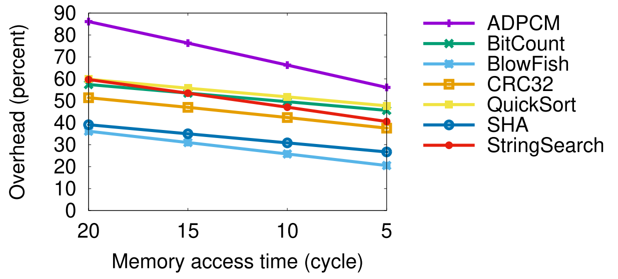

FIGURE 3. The overhead of proof generation when varying the memory access time. The access time is measured in processor clock cycles. The cache size is 16 KB.

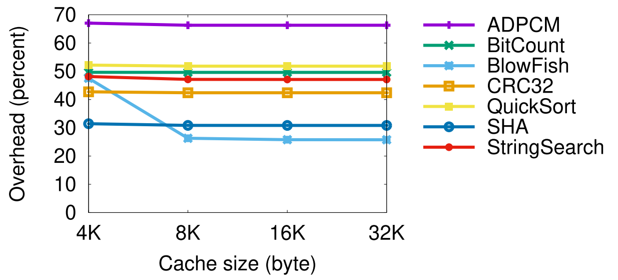

FIGURE 4. The overhead of proof generation when varying the cache size. The memory access time is 10 cycles.

the perceived memory access time increases. Thus, we measure the performance overhead by varying (a) the memory access time, and (b) the cache size.

The main memory is typically implemented as Dynamic Random Access Memory (DRAM). The access time of DRAM heavily depends on the access patterns. Since the access patterns are not the focus of this work, we assume the use of Static Random Access Memory (SRAM) as the main memory in this experiment to eliminate unexpected artifacts on the performance. It always takes the same number of clock cycles to access the memory.

The hash algorithm does not have a direct impact on the performance overhead. The overhead is incurred because of the memory accesses. Once the data is fetched, the hash computation is done by dedicated computation logic, which does not interfere with the execution of the program. In other words, the computation time of the hash is not visible, because it is performed in parallel with the program execution. In contrast, the overhead of signature generation is visible, because it can only be performed after the program completes.

Figures 3 and 4 show the results. As expected, the longer the memory access time is, the more overhead is incurred. Similarly, the smallest cache size incurs the most overhead.

The overhead is at most 86.12% in the case of a 20-cycle memory access time and a 16 KB cache size. Figure 5 shows the breakdown of the computation time for these particular parameters. On average, 13.74% of the overhead is attributed to the interference with memory accesses, and 21.43% to the signature generation.

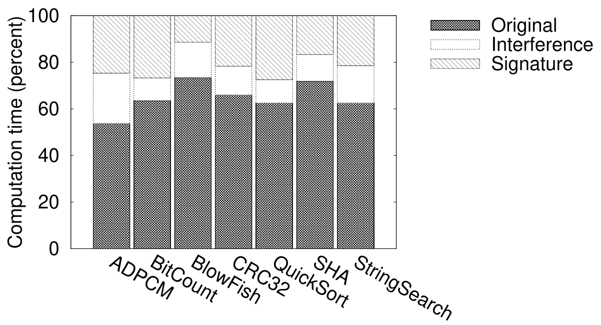

FIGURE 5. Breakdown of the computation time. 'Original' refers to the original computation time, i.e., the time taken to execute the program without computing the hash and signature. 'Interference' refers to the increased time due to the interference with memory accesses. 'Signature' refers to the time taken to generate the signature.

TABLE 4. Comparison of the execution time with previous work. 'Original' is included as a reference, which is the execution time of the benchmark without computing the hash and signature. Note that the time unit of previous work is hours, whereas the time unit of the proposed scheme is milliseconds.

Benchmark	Original	Proposed	No-limit [32]	Limit [10]
ADPCM	1.82 ms	2.01 ms	885.50 h	3.00 h
BitCount	1.99 ms	2.06 ms	843.45 h	2.86 h
BlowFish	5.41 ms	5.68 ms	2,713.59 h	N/A
CRC32	2.55 ms	2.67 ms	1,000.40 h	3.39 h
QuickSort	1.91 ms	1.98 ms	922.18 h	3.13 h
SHA	3.62 ms	3.75 ms	1,745.45 h	6.58 h
StringSearch	2.45 ms	2.60 ms	1,219.38 h	4.60 h

To compare the prover overhead with prior techniques, we translate the numbers of cycles to actual times by assuming the processor runs at 100 MHz (10 ns/cycle). We assume this clock speed, because the maximum clock speed of the ECDSA logic is 107.4 MHz [43]. Since we assume a low clock frequency, a 5-cycle memory access time is used for the comparison. Table 4 shows a comparison of the execution time of the benchmarks – when verifiable computation is enabled – between the proposed scheme and two state-of-the-art software-based related works [10], [32].

The references report the execution time of one instruction. The overall execution time depends on the program size and the input size. We pick the execution time from the table in each reference (according to the program size and the input size of each application), and multiply it by the number of executed instructions. Specifically, Table 5 shows precisely how the execution time is calculated. The execution time per instruction ("Per inst. (β)" in the table) is taken from reference [32] according to the program size and the input size. In the same way, γ is taken from reference [10]. The number of executed instructions ("No. of executed inst. (α)" in the table) is obtained from our simulation framework. The total execution time ("Exe. time" in the table) is calculated by multiplying the execution time per instruction with the number of executed instructions.

The two previous works [10], [32] support programs running on general-purpose Von Neumann processors. While the work in [32] has no limitation on the number of executed instructions, the other work [10] imposes a limit. The overhead of [10] depends on the limit set on the number of executed instructions. Even if the technique in [10] imposes a limit on the number of instructions, it is significantly faster than the work without limitation [32]. In the case of the BlowFish benchmark, the execution time is not available for [10], because it exceeds the aforementioned limit.

As clearly shown in the table, both previous works incur significantly higher performance overhead, because they verify every instruction fetch, instruction execution, and memory access. Furthermore, the overhead of the key generation in the previous works is not even included in the table. On the other hand, the proposed scheme drastically reduces the overhead by introducing a CEE, which allows for the omission of the verification of program execution.

B. VERIFIER OVERHEAD

When the verifier receives the output and the proof, it verifies them by itself computing the proof. The hashes of S_0^V can be computed before the verifier receives the output, or they may be publicly available. However, in this experiment, we include the hash computation to the verification time. The verification time is measured on the same AMBER processor with a 5-cycle memory access time and 16 KB cache. A hardware accelerator for the hash and signature verification is assumed [43].

The verification time results are shown in Table 6. As a reference, we compare the verification time of the benchmarks against their original execution time. If verifiable computation is used to outsource the computation, the verification time should be shorter than the original execution time. Indeed, as shown in Table 6, the verification time for the proposed approach is always shorter than the original execution time. The verification time is at most 78.69% of the original execution time (in the case of the ADPCM benchmark). On average, the verification time is 58.50% of the original execution time. The verification time of the previous works is linearly related to the program size (No-limit) and the input size (Limit). The relation between the program size and the verification time is given in reference [32]: verification time $= 23.78 + 0.00702 \times$ program size (ms). We use this formula to calculate the verification. However, in the case of "Limit", the formula is not given. Thus, we derive our own formula, by regression, based on the results reported in reference [10]: $40.98 + 0.00075 \times$ input size (ms).

Existing verifiable computation schemes require evaluation and verification keys of large sizes. In our scheme, the private key of ECDSA is used as an evaluation key, and the public key of ECDSA as a verification key. Table 7 compares the key sizes. The evaluation key of the work in [10] (which is denoted by "Limit" in the table) varies with the number of executed instructions. The key size of the ECDSA hardware accelerator we used in our experiments is 163 bits, as recommended by NIST [43].

TABLE 5. Execution time of previous cryptographic approaches.

Application	Program size	Input size	No. of executed inst. (α)	No-limit [32]		Limit [10]	
				Per inst. (β)	Exe. time ($\alpha \times \beta$)	Per inst. (γ)	Exe. time ($\alpha \times \gamma$)
ADPCM	5,116 B	3,072 B	121,672	26,200 ms	885.50 h	89.0 ms	3.00 h
BitCount	4,828 B	292 B	115,895	26,200 ms	843.45 h	89.0 ms	2.86 h
BlowFish	4,820 B	4,258 B	372,860	26,200 ms	2713.59 h	Exceeds the limit	
CRC32	4,576 B	1,136 B	137,460	26,200 ms	1000.40 h	89.0 ms	3.39 h
QuickSort	4,320 B	528 B	126,712	26,200 ms	922.18 h	89.0 ms	3.13 h
SHA	5,968 B	672 B	239,833	26,200 ms	1745.45 h	98.9 ms	6.58 h
SHA	4,444 B	2,852 B	167,549	26,200 ms	1219.38 h	98.9 ms	4.60 h

TABLE 6. Comparison of the verification times with previous work. 'Original' is included as a reference, which is the execution time of the benchmark without computing the hash and signature.

Benchmark	Original	Proposed	No-limit [32]	Limit [10]
ADPCM	1.82 ms	1.43 ms	59.69 ms	43.26 ms
BitCount	1.99 ms	1.56 ms	57.67 ms	41.19 ms
BlowFish	5.41 ms	1.52 ms	57.61 ms	N/A
CRC32	2.55 ms	1.73 ms	55.90 ms	44.14 ms
QuickSort	1.91 ms	1.24 ms	54.10 ms	41.37 ms
SHA	3.62 ms	1.30 ms	65.67 ms	41.48 ms
StringSearch	2.45 ms	1.35 ms	54.97 ms	43.09 ms

TABLE 7. Comparison of evaluation and verification key sizes.

	Proposed	No-limit [32]	Limit [10]
Evaluation key	41 B	55 MB	59 - 122 TB
Verification key	21 B	1.3 KB	17 KB

C. COMPARISON WITH INTEL SGX-BASED PROTOTYPE

We compare the performance overhead of our light-weight CEE prototype with that of the Intel SGX-based CEE prototype. We measure the ratio of the prover execution time over the original execution time. The second column of Table 8 (titled 'LW CEE') shows the ratio of our light-weight prototype, which is the ratio of the third column over the second column of Table 4. The third column of Table 8 (titled 'SGX CEE') shows the ratio of the Intel SGX-based CEE prototype. In fact, most of the overhead is caused by the hash and signature computation, because these are implemented in software running in Enclave. The relative overhead may look excessive, but this is due to the small benchmarks that were used. Note, again, that we had to limit the benchmark size, in order to enable comparison with previous works [10], [32], since those works impose limitations. Regardless, even if we exclude the hash and signature computation, Intel's SGX still incurs significant overhead. The fourth column of Table 8 (titled 'SGX w/o VC') shows the ratio without the hash and signature computation. Specifically, it is the time taken to execute the benchmark in Enclave over the time taken by native execution. All measurement results with Enclave for this experiment do not include the initialization time of Enclave. If they did, the overhead of the SGX-based prototype would be even higher. Nevertheless, these experimental results clearly demonstrate that the overhead of our light-weight CEE prototype is markedly lower than that of the Intel SGX-based CEE prototype.

TABLE 8. The ratio of the prover execution time over the original execution time.

Benchmark	LW CEE	SGX CEE	SGX w/o VC
ADPCM	1.10	41.89	1.78
BitCount	1.03	609.00	42.00
BlowFish	1.05	6.66	1.30
CRC32	1.04	200.66	8.66
QuickSort	1.03	306.33	9.33
SHA	1.03	63.80	3.70
StringSearch	1.06	226.33	69.00

D. HARDWARE COST

To implement the hardware-based CEE, the AMBER processor is modified. The additional logic for the interface, state preservation, hash computation, and signature generation increases the hardware cost. The area of the hardware is compared with and without modification.

Table 9 shows the comparison of the occupied hardware area. The hardware area is measured using the Nangate 45nm open cell library. The area overhead to implement CEE is 41.47% and 19.86% in modifying the register file and execution stage, respectively. For verifiable computation, the cache controller is modified to compute the hash, which incurs 209.71% area overhead. The 'Signature' component is the computation logic for SHA and ECDSA. The signature generation logic is the biggest component in our prototype. However, we regard the *performance* overhead as the top priority in our prototype implementation. Considering the trade-off between the performance and the hardware cost, we may consider other implementations to reduce the hardware cost, if necessary. The interface logic is also added, which occupies 0.004 mm^2. The area of the unchanged logic is 0.085 mm^2.

E. SECURITY VALIDATION

We validate that the proposed scheme successfully thwarts the following attack scenarios:

- A malicious prover sends an output which is not generated by the CEE. Since the prover cannot access the private key of the CEE, the signature cannot be counterfeited. Thus, the *Verify* function generates 0, and the verifier notices that the output is not correct.

TABLE 9. Hardware cost analysis. The area of RAM is not included.

Component	Original	Modified	Overhead
Register file	0.023 mm^2	0.032 mm^2	41.47 %
Execution stage	0.021 mm^2	0.025 mm^2	19.86 %
Cache controller	0.002 mm^2	0.006 mm^2	209.71 %
Signature	-	0.257 mm^2	-
Interface	-	0.004 mm^2	-
Unchanged	0.085 mm^2	0.085 mm^2	-
Total	0.131 mm^2	0.409 mm^2	213.72 %

- A malicious prover provides program code and input to the CEE, which are not received by the verifier. Since the CEE generates hashes for the code and input, they cannot be matched with what the verifier expects. In other words, the *Verify* function generates 0.
- A malicious prover provides a memory layout that is different from what the verifier sends. Since the CEE generates a hash on the memory layout as well, if this happens the signature mismatches. The verifier can notice this, because the *Verify* function generates 0.
- While the protected program is being executed, it may be scheduled out and another program can be scheduled in. We execute a malicious program that modifies register values before the external function returns. The protected program on the CEE works correctly, because all register values are restored when the program resumes.
- We test another malicious program that tries to modify the program code, input, and output. In this case, an exception is raised, and the malicious program is terminated, because the hardware detects violation. Even if this happens, the protected program generates the correct output because the hardware prevents the attack.

The proposed scheme cannot defend against the following cases:

- Adversaries may modify the memory and registers directly by using physical attacks. The CEE prevents illegal modifications to the memory by preventing store instructions, and it prevents modifications to the registers by saving them inside the processor. However, if adversaries use other means (e.g., debugging interface) to modify the memory and registers, the CEE cannot prevent such scenarios. If the attackers can modify the memory and registers, they can change the instructions, input, and output and deceive the verifier.
- Adversaries may extract the private key from the hardware through physical attacks. Through software means, adversaries cannot extract the private key, because the hardware does not provide any interface for it. However, adversaries may use special equipment to read the private key without using a software interface.
- If the program is developed in an unexpected (not according to specifications) way, correct output cannot be guaranteed. The CEE has minimal capability to check the behavior of the program (i.e., the CEE checks whether the program reads data only from R and T), but it does not guarantee correctness. For example, if the final output is not written to W, its hash cannot be generated by the CEE. This is the same situation as in the case where the program has a bug. As previously stated, it is ultimately the program developer's responsibility to develop the program in a correct way.

F. SUMMARY

Table 10 summarizes the attributes of different VC schemes. Most of the existing crytographic approaches [12] only support arithmetic operations, while some recent works [10], [32] support C-like programs. The SGX-based approach and the proposed technique support programs to be outsourced. As demonstrated through experiments, the execution time of cryptographic approaches is very long and requires large keys. The SGX-based approach offers better execution time, while the proposed technique achieves even better execution time by supporting essential functionalities for VC. However, both the SGX-based and the proposed approaches require trusted hardware, whereas the cryptographic approaches rely only on cryptographic premises.

VII. RELATED WORK

This section reviews existing verifiable computation schemes. Current verifiable computation schemes are mostly based on cryptographic algorithms (computation complexity theory). We hereby discuss these techniques in detail in the following sub-section (Section VII-A), followed by hardware-based approaches in Section VII-B.

A. CRYPTOGRAPHIC APPROACHES

Extensive research has been conducted on verifiable computation in cryptographic algorithms. It originates from Interactive Proofs [46], which are based on Probabilistically Checkable Proofs (PCP). Although PCP provides a user with the ability to check and verify the data processing result, it is not practical, since it requires a huge amount of data locally, and the verification demands large computations.

To improve PCP, a linear PCP has been introduced [47], [48]. The linear PCP constructs a proof by using only linear operations. A prover generates a commitment using a linearly homomorphic function, and then a verifier checks the commitment efficiently. Although the linear PCP schemes are more efficient than PCP schemes, they still have the drawbacks of large-size keys and slow proving time.

To utilize a tradeoff between proof size, proof time, and verification time, different trust setup models and cryptographic assumptions are adopted. The most efficient model in terms of proof size and verification time is a preprocessing model, whereby a one-time setup is required to generate an Evaluation Key (EK) and a Verification Key (VK), which are called a Structured Reference String (SRS). Gennaro *et al.* [11] proposes a pairing-based scheme with Quadratic Span Program (QSP), in which a circuit is composed of the two boolean operations of addition and

TABLE 10. Attributes of existing and the proposed VC schemes.

	Cryptographic approaches		Trusted hardware approaches	
	Arithmetic [12]	Program [32], [10]	SGX-based [25]	Proposed
Outsourced computation	Arithmetic operation	Program	Program	Program
Execution time	Extremely long	Extremely long	Long	Short
Key size	Large	Large	Small	Small
Root of trust	Cryptography	Cryptography	Trusted hardware	Trusted hardware

multiplication. In this scheme, the key size and the proving time are linearly proportional to the circuit size, while the verification time and the proof size remain constant. The technique has been extended in [12], [36], [49]–[52]. Unfortunately, these systems require a trust setup, in which a key should be generated by a trust party who knows a trapdoor value that should not be revealed to provide soundness. In addition, a key should be generated according to a given circuit.

Recently, updatable SRS schemes have been introduced to allow for the updating of EK and VK [53]–[55], where null vector computation and a polynomial commitment are utilized. In the updatable SRS schemes, a universal circuit-independent SRS is generated and can be updated by any user multiple times. The universal SRS allows the generation of a proof for any circuit without regenerating an SRS for the given circuit. Moreover, as long as there exists at least one honest user involved in the SRS update, the soundness is preserved.

If no trusted setup is required, it is called transparent [7], [8], [56]–[58]. Bulletproof [8] efficiently reduces the proof size logarithmically with the circuit size, by optimizing inner product operations. However, the verification time scales linearly. Hyrax [7] is based on a sum-check protocol introduced by Goldwasser *et al.* [46], and improved by [5]. Dark [56] is an efficient polynomial commitment scheme using an unknown order group. It is the first approach to generate a logarithmic-size proof and logarithmic verification time. STARKs, which uses a hash function, is a post-quantum resistant scheme that minimizes proof size ($O(\log^2(n))$) and verification time ($O(\log^2(n))$), where n denotes the circuit size.

To utilize cryptographic verifiable computation, many front-end tools have been developed [10], [32], [36], [51], [59]–[66]. The authors of [10], [32], [51] construct a tinyRAM-based processor circuit, and a tinyRAM machine code is provided to the processor. The work in [66] proposes a tool to generate R1CS (Rank-1 Constraint System) code for back-end cryptographic approaches from Java-like high-level language code. Pinocchio [36], Geppetto [61], and Pequin [64] accept C (or a subset of C) code as input program.

B. HARDWARE APPROACHES

Trusted hardware has been extensively studied as a means to enhancing security. Since it is much harder to modify hardware than software, hardware implementation of security properties can offer strong invariants.

Existing trusted hardware approaches can be classified into hardware-based root-of-trust [17], [18], attestation [19]–[21], and isolation [22]–[24]. In the hardware-based root-of-trust approaches [17], [18], the security of a system is enhanced by building the system based on the invariants guaranteed by dedicated hardware. For example, the private key stored in Trusted Platform Module (TPM) [18] can be used to authenticate software running on the system. The attestation techniques [19]–[21], [23], [26]–[28] are used to validate the integrity of a program, its execution (control flow integrity), and/or data integrity. The isolation approaches [22]–[24] provide a hardware mechanism to segregate system resources from potentially malicious programs. None of this specialized hardware is designed for VC, and, consequently, none of it can guarantee (by itself) the requirements for VC. Intel's SGX incorporates all of these, but it is still not enough, because the registers should also be preserved.

There are, however, approaches that employ SGX for VC of a specific application domain. Specifically, the SGX scheme has been used to accelerate VC for smart contracts [67], machine learning [68], secure function evaluation [69], and MapReduce [70], [71]. Verifiable ASICs [72] employ a hardware circuit to verify the untrusted hardware modules. These techniques target a specific domain of applications, and they are not geared for general-purpose use.

To employ trusted hardware for general-purpose verifiable computation, we need to define the requirements that the hardware should guarantee. To the best of our knowledge, however, there has been no prior work that theoretically defines the requirements. The attestation technique of SGX has been analyzed mathematically [30], but it does not cover verifiable computation. As previously mentioned, the five essential building blocks of a Trusted Execution Environment (TEE) are identified in [31], but they are also not geared for verifiable computation. In this paper, we formally define the notion of a CEE for verifiable computation, and discuss how to realize it.

A CEE can be implemented using SGX. Sealed-Glass Proof (SGP) [29] proposes the concept of transparent SGX, which guarantees integrity and authenticity of program execution, but not confidentiality. Transparent SGX can be used as a CEE, if OCALL is not allowed, as previously explained in Section III-C.

VIII. CONCLUSION

In this paper, we propose a practical verifiable computation scheme enabled by a hardware-based Correct Execution Environment (CEE). The CEE guarantees that the correct output is always generated by a protected program, even if other programs that are concurrently running on the same platform are compromised. Malicious programs are prevented by the CEE from distorting the output of the protected program. This is achieved through the preservation of the register values and the blocking of any modifications to the memory regions used by the protected program. A signature is generated by the hardware-based CEE, so that the verifier can verify the authenticity of the input, output, and the program. By employing a CEE, the overhead of verifiable computation is drastically reduced, as compared to existing cryptography-based approaches, as demonstrated by various experiments.

REFERENCES

[1] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister, "Verifiable computation with massively parallel interactive proofs," in *Proc. USENIX Workshop Hot Topics Cloud Comput.*, 2012, pp. 1–22.

[2] D. C. Sánchez, "Raziel: Private and verifiable smart contracts on blockchains," *CoRR*, vol. abs/1807.09484, 2018. [Online]. Available: http://arxiv.org/abs/1807.09484

[3] L. P. Maddali, M. S. D. Thakur, R. Vigneswaran, M. A. Rajan, S. Kanchanapalli, and B. Das, "VeriBlock: A novel blockchain framework based on verifiable computing and trusted execution environment," in *Proc. Int. Conf. Commun. Syst. Netw. (COMSNETS)*, Jan. 2020, pp. 1–6.

[4] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: Interactive proofs for muggles," in *Proc. 14th Annu. ACM Symp. Theory Comput.*, 2008, pp. 113–122.

[5] G. Cormode, M. Mitzenmacher, and J. Thaler, "Practical verified computation with streaming interactive proofs," in *Proc. 3rd Innov. Theor. Comput. Sci. Conf. (ITCS)*, 2012, pp. 90–112.

[6] R. S. Wahby, Y. Ji, A. J. Blumberg, A. Shelat, J. Thaler, M. Walfish, and T. Wies, "Full accounting for verifiable outsourcing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2071–2086.

[7] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish, "Doubly-efficient zkSNARKs without trusted setup," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Francisco, CA, USA, 2018, pp. 926–943.

[8] B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 315–334.

[9] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "VRAM: Faster verifiable RAM with program-independent preprocessing," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 908–925.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proc. 23rd USENIX Conf. Secur. Symp.* Berkeley, CA, USA: USENIX Association, 2014, pp. 781–796.

[11] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct NIZKs without PCPs," in *Proc. 32nd Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Athens, Greece, 2013, pp. 626–645.

[12] J. Groth, "On the size of pairing-based non-interactive arguments," in *Proc. 35th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Vienna, Austria, 2016, pp. 305–326.

[13] I. Giacomelli, J. Madsen, and C. Orlandi, "Zkboo: Faster zero-knowledge for Boolean circuits," in *Proc. USENIX Secur. Symp.*, 2016, pp. 1069–1083.

[14] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs," in *Proc. 37th Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, 2017, pp. 581–612.

[15] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2014, pp. 459–474, doi: 10.1109/SP.2014.36.

[16] J. Eberhardt and S. Tai, "ZoKrates–Scalable privacy-preserving off-chain computations," in *Proc. IEEE Int. Conf. Internet Things (iThings) IEEE Green Comput. Commun. (GreenCom) IEEE Cyber, Phys. Social Comput. (CPSCom) IEEE Smart Data (SmartData)*, Halifax, NS, Canada, Jul. 2018, pp. 1084–1091.

[17] B. Parno, J. M. McCune, and A. Perrig, "Bootstrapping trust in commodity computers," in *Proc. IEEE Symp. Secur. Privacy*, May 2010, pp. 414–429.

[18] Trusted Computing Group. *Trusted Platform Module*. Accessed: 2011. [Online]. Available: https://trustedcomputinggroup.org

[19] M. Geden and K. Rasmussen, "Hardware-assisted remote runtime attestation for critical embedded systems," in *Proc. 17th Int. Conf. Privacy, Secur. Trust (PST)*, Aug. 2019, pp. 1–10.

[20] I. D. O. Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik, "VRASED: A verified hardware/software co-design for remote attestation," in *Proc. 28th USENIX Secur. Symp.*, Aug. 2019, pp. 1429–1446.

[21] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices," in *Proc. IEEE Symp. Secur. Privacy*, May 2004, pp. 272–282.

[22] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: hardware-assisted data-flow isolation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 1–17.

[23] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune, "Memoir: Practical state continuity for protected modules," in *Proc. IEEE Symp. Secur. Privacy*, May 2011, pp. 379–394.

[24] J. Jang, C. Choi, J. Lee, N. Kwak, S. Lee, Y. Choi, and B. B. Kang, "PrivateZone: Providing a private execution environment using ARM TrustZone," *IEEE Trans. Dependable Secure Comput.*, vol. 15, no. 5, pp. 797–810, Sep. 2018.

[25] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[26] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-overhead control flow attestation in hardware," in *Proc. 54th Annu. Design Autom. Conf.* New York, NY, USA: Association Computing Machinery, Jun. 2017, pp. 1–6.

[27] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 263–277.

[28] S. A. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Apr. 2017, pp. 193–204.

[29] F. Tramer, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Apr. 2017, pp. 19–34.

[30] M. Barbosa, B. Portela, G. Scerri, and B. Warinschi, "Foundations of hardware-based attested computation and application to SGX," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 14, 2016.

[31] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *Proc. IEEE Trustcom/BigDataSE/ISPA*, vol. 1, Aug. 2015, pp. 57–64.

[32] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," *Algorithmica*, vol. 79, no. 4, pp. 1102–1160, Dec. 2017.

[33] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," in *Proc. 21st Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2016, pp. 743–755.

[34] K. Rosenfeld and R. Karri, "Attacks and defenses for JTAG," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 27, no. 1, pp. 36–47, Jan. 2010.

[35] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. 2017 USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 645–658.

[36] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, Berkeley, CA, USA, May 2013, pp. 238–252, doi: 10.1109/SP.2013.47.

[37] A. Kosba, C. Papamanthou, and E. Shi, "XJsnark: A framework for efficient verifiable computation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 944–961.

[38] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2014, pp. 1–6.

[39] C. Santifort. (2015). *Amber Core 2 Specication*. [Online]. Available: https://opencores.org/projects/amber
[40] M. Imran, M. Rashid, A. R. Jafri, and M. Najam-ul-Islam, "ACryp-Proc: Flexible asymmetric crypto processor for point multiplication," *IEEE Access*, vol. 6, pp. 22778–22793, 2018.
[41] M. Rashid, M. Imran, A. R. Jafri, and T. F. Al-Somani, "Erratum: Flexible architectures for cryptographic algorithms—A systematic literature review," *J. Circuits, Syst. Comput.*, vol. 28, no. 13, Dec. 2019, Art. no. 1992002.
[42] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proc. 4th Annu. IEEE Int. Workshop Workload Characterization*, Dec. 2001, pp. 3–14.
[43] A. Sghaier, M. Zeghid, and M. Machhout, "Fast hardware implementation of ECDSA signature scheme," in *Proc. Int. Symp. Signal, Image, Video Commun. (ISIVC)*, Dec. 2016, pp. 343–348.
[44] H. Handschuh, *SHA-0, SHA-1, SHA-2 (Secure Hash Algorithm)*. Boston, MA, USA: Springer, 2011, pp. 1190–1193.
[45] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ECDSA)," *Int. J. Inf. Secur.*, vol. 1, no. 1, pp. 36–63, Aug. 2001.
[46] S. Goldwasser, Y. T. Kalai, and N. G. Rothblum, "Delegating computation: Interactive proofs for muggles," *J. ACM*, vol. 62, no. 4, pp. 27:1–27:64, Sep. 2015.
[47] S. T. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)," in *Proc. NDSS*, 2012, vol. 1, no. 9, p. 17.
[48] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish, "Taking proof-based verified computation a few steps closer to practicality," in *Proc. 21st {USENIX} Secur. Symp.*, 2012, pp. 253–268.
[49] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish, "Resolving the conflict between generality and plausibility in verified computation," in *Proc. 8th ACM Eur. Conf. Comput. Syst. (EuroSys)*, 2013, pp. 71–84.
[50] N. Bitansky, A. Chiesa, Y. Ishai, O. Paneth, and R. Ostrovsky, "Succinct non-interactive arguments via linear interactive proofs," in *Proc. Theory Cryptogr. Conf.* Boston, MA, USA: Springer, 2013, pp. 315–333.
[51] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for C: Verifying program executions succinctly and in zero knowledge," in *Advances in Cryptology–CRYPTO 2013*. Boston, MA, USA: Springer, 2013, pp. 90–108.
[52] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou, "VSQL: Verifying arbitrary SQL queries over dynamic outsourced databases," in *Proc. IEEE Symp. Secur. Privacy (SP)*. San Jose, CA, USA: IEEE Computer Society, May 2017, pp. 863–880.
[53] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers, "Updatable and universal common reference strings with applications to zk-SNARKs," in *Proc. 38th Annu. Int. Cryptol. Conf.*, Santa Barbara, CA, USA, 2018, pp. 698–728, doi: 10.1007/978-3-319-96878-0_24.
[54] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, London, U.K., Nov. 2019, pp. 2111–2128, doi: 10.1145/3319535.3339817.
[55] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge," Cryptol. ePrint Arch., Lyon, France, Tech. Rep. 2019/953, 2019.
[56] B. Bünz, B. Fisch, and A. Szepieniec, "Transparent snarks from DARK compilers," in *Proc. 39th Annu. Int. Conf. Theory Appl. Cryptograph. Techn.*, Zagreb, Croatia, 2020, pp. 677–706, doi: 10.1007/978-3-030-45721-1_24.
[57] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable zero knowledge with no trusted setup," in *Proc. Annu. Int. Cryptol. Conf.* Boston, MA, USA: Springer, 2019, pp. 701–732.
[58] E. Ben-Sasson, A. Chiesa, M. Riabzev, N. Spooner, M. Virza, and N. P. Ward, "Aurora: Transparent succinct arguments for R1CS," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Boston, MA, USA: Springer, 2019, pp. 103–128.
[59] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "ZKPDL: A language-based system for efficient zero-knowledge proofs and electronic cash," in *Proc. USENIX Secur. Symp.*, vol. 10, 2010, pp. 193–206.
[60] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish, "Verifying computations with state," in *Proc. 24th ACM Symp. Operating Syst. Princ. SOSP*, 2013, pp. 341–357.
[61] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 253–270.
[62] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish, "Efficient RAM and control flow in verifiable outsourced computation," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2015, pp. 1–15.
[63] A. Kosba, Z. Zhao, A. Miller, Y. Qian, H. Chan, C. Papaman-Thou, R. Pass, S. Abhi, and E. Shi, "C0c0: A framework for building composable zero-knowledge proofs," Cryptol. ePrint Arch., Lyon, France, Tech. Rep. 2015/1093, 2015. [Online]. Available: http://eprint.iacr.org
[64] Pepper Project. (2016). *Pequin: An End-To-End Toolchain for Verifiable Computation, Snarks, and Probabilistic Proofs*. [Online]. Available: https://github.com/pepper-project/pequin
[65] G. Stewart, S. Merten, and L. Leland, "Snårkl: Somewhat practical, pretty much declarative verifiable computing in Haskell," in *Proc. Int. Symp. Practical Aspects Declarative Lang.* Boston, MA, USA: Springer, 2018, pp. 36–52.
[66] A. Kosba, C. Papamanthou, and E. Shi, "XJsnark: A framework for efficient verifiable computation," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 944–961.
[67] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contract execution," *CoRR*, vol. abs/1804.05141, 2018. [Online]. Available: http://arxiv.org/abs/1804.05141
[68] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 619–636.
[69] J. I. Choi, D. Tian, G. Hernandez, C. Patton, B. Mood, T. Shrimpton, K. R. B. Butler, and P. Traynor, "A hybrid approach to secure function evaluation using SGX," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2019, pp. 100–113.
[70] Y. Wang, Y. Shen, and X. Jiang, "Practical verifiable Computation—A MapReduce case study," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 6, pp. 1376–1391, Jun. 2018.
[71] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 38–54.
[72] R. S. Wahby, M. Howald, S. Garg, A. Shelat, and M. Walfish, "Verifiable ASICs," in *Proc. IEEE Symp. Secur. Privacy (SP)*, San Jose, CA, USA, May 2016, pp. 759–778.

JUNGHEE LEE (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, in 2000 and 2003, respectively, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, in 2013. He is currently with the School of Cybersecurity, Korea University, since 2019. From 2003 to 2008, he worked at Samsung Electronics on electronic system level design of mobile system-on-chip. From 2014 to 2019, he was with the Department of Electrical and Computer Engineering, University of Texas at San Antonio, as an Assistant Professor. His research interests include secure design or hardware-assisted security of processor, non-volatile memory, storage, and dedicated hardware.

CHRYSOSTOMOS NICOPOULOS (Member, IEEE) received the B.S. and Ph.D. degrees in electrical engineering with a specialization in computer engineering from Pennsylvania State University, State College, PA, USA, in 2003 and 2007, respectively. From 2007 to 2008, he worked as a Postdoctoral Research Associate with the Processor Architecture Laboratory, Ecole Polytechnique Federale de Lausanne (EPFL), Switzerland. He is currently an Associate Professor with the Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus, where he leads the multicore Computer Architecture Laboratory (multiCAL). His research interests include computer architecture, microprocessor and computer system design, and networks-on-chip.

GWEONHO JEONG received the B.S. degree in information systems engineering from Hanyang University, Seoul, Korea, where he is currently pursuing the Ph.D. degree in information systems engineering. His current research interests focus on applied cryptography including protocols for forward secure signatures, PKE, and zero-knowledge proofs.

JIHYE KIM (Member, IEEE) received the B.S. and M.S. degrees from the School of Computer Science and Engineering, Seoul National University, South Korea, in 1999 and 2003, respectively, and the Ph.D. degree in computer science from the University of California, Irvine, in 2008. She is currently an Assistant Professor with the Department of Electrical Engineering, Kookmin University. Her research interests include applied cryptography, verifiable computation, and zero-knowledge proofs.

HYUNOK OH (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1996, 1998, and 2003, respectively. He is currently a Full Professor with the Department of Information Systems, Hanyang University, Seoul. His research interests include applied cryptography, zero-knowledge proofs, verifiable computation, non-volatile memory, and embedded systems.