IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received 25 July 2024, accepted 11 August 2024, date of publication 14 August 2024, date of current version 26 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3443509

RESEARCH ARTICLE

A Hardware-Based Correct Execution Environment Supporting Virtual Memory

DAEHYEON LEE[1], OHSUK SHIN[1], YEONGHYEON CHA[1], JUNGHEE LEE[1], (Member, IEEE), TAISIC YUN[2], JIHYE KIM[3], (Member, IEEE), HYUNOK OH[4], (Member, IEEE), CHRYSOSTOMOS NICOPOULOS[5], (Member, IEEE), AND SANG SU LEE[6]

[1]School of Cybersecurity, Korea University, Seoul 02841, South Korea
[2]Graduate School of Information Security, Korea Advanced Institute of Science and Technology, Daejeon 34141, South Korea
[3]Electronics and Information System Engineering Major, Kookmin University, Seoul 02707, South Korea
[4]Department of Information System, Hanyang University, Seoul 04763, South Korea
[5]Department of Electrical and Computer Engineering, University of Cyprus, 1678 Nicosia, Cyprus
[6]Cyber Security Research Division, ETRI, Daejeon 34129, South Korea

Corresponding author: Junghee Lee (j_lee@korea.ac.kr)

This research was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government [Ministry of Science and Information and Communication Technologies (MSIT)] (RS-2021-II210528, Development of Hardware-centric Trusted Computing Base and Standard Protocol for Distributed Secure Data Box, 50%) (RS-2020-II200215, The Development of H/W Vulnerability Assessment Technologies against Supply-Chain Attacks, 50%).

ABSTRACT The rapid increase in data generation has led to outsourcing computation to cloud service providers, allowing clients to handle large tasks without investing resources. However, this brings up security concerns, and while there are solutions like fully homomorphic encryption and specific task-oriented methods, challenges in optimizing performance and enhancing security models remain for widespread industry adoption. Outsourcing computations to an untrusted remote computer can be risky, but attestation techniques and verifiable computation schemes aim to ensure the correct execution of outsourced computations. Nevertheless, the latter approach incurs significant overhead in generating a proof for the client. To minimize this overhead, the concept of a Correct Execution Environment (CEE) has been proposed (CEEv1), which omits proof generation for trusted parts of the prover. This paper proposes a new hardware-based CEE (CEEv2) that supports virtual memory and uses an inverted page table mechanism to detect, or prevent, illegal modifications to page mappings. The proposed mechanism supports virtual memory and thwarts virtual-to-physical mapping attacks, while minimizing software modifications. The paper also compares the proposed mechanism to other similar mechanisms used in AMD's SEV-SNP and Intel's SGX.

INDEX TERMS Hardware, verifiable computation, isolation.

I. INTRODUCTION

Outsourcing computations to an untrusted remote computer is a challenging problem [2]. Attestation techniques aim at guaranteeing the correct execution of the outsourced computations by guaranteeing the integrity of the code, control flow, and data. To avoid Time-Of-Check Time-Of-Use (TOCTOU) attacks, an isolation technique is often employed along with the attestation to preserve the integrity during the execution [3].

The rapid increase in data generation has heightened interest in outsourcing computation to cloud service providers, allowing resource-limited clients to manage large-scale tasks without heavy investment in hardware. However, this raises security and privacy concerns when sensitive data is processed in potentially untrusted cloud environments. Secure computation outsourcing systems typically involve the client, the cloud server, and sometimes an independent verifier to

The associate editor coordinating the review of this manuscript and approving it for publication was Kashif Saleem.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

ensure result correctness. Solutions are categorized into general solutions, like fully homomorphic encryption (FHE) for high-security but computationally heavy tasks, and specific tasks like matrix operations, systems of linear equations (SLE), and mathematical optimization problems. Despite progress, challenges remain in optimizing performance, enhancing security models, and creating scalable, practical implementations for widespread industry adoption [4].

Verifiable computation schemes aim at the same goal, but with a different methodology. In said schemes, the server (called prover in verifiable computation) returns the computation result with a *proof* to the client (called verifier), so that the verifier can check the correctness of the result. This is achieved by employing cryptographic algorithms and the scheme guarantees formally proven security properties [5], [6].

However, the challenge is that a significant overhead is incurred in generating a proof. Compared to native execution, it takes 10,000 to 100,000 times longer to generate a proof while executing the outsourced computation [7]. To reduce such an excessive overhead, the concept of Correct Execution Environment (CEE) has been proposed [7] (CEEv1). Its key idea is that we can omit the generation of a proof for a part of the prover, if we can trust that part. CEEv1 guarantees two security properties, soundness and completeness, which are formally defined and proved [7]. The attributes that the trusted part must guarantee were formally defined and a hardware prototype of the CEEv1 was presented. Nevertheless, the focus of said work [7] was on the *theoretical* foundation of the CEEv1, not on the hardware implementation. A proof-of-concept prototype was only presented to demonstrate the feasibility of a hardware implementation. While that original prototype worked well for low-end devices, it did not support virtual memory. Since computations are likely to be outsourced to a server that operates using virtual memory, then support for virtual memory is an essential capability for wide deployment of the CEEv1. It should be noted that the original hardware CEEv1 implementation had the advantage that only minimal modification to the software was required. This attribute is instrumental in facilitating widespread adoption of the CEEv1 and should – ideally – be maintained in any future CEEv1 implementations.

In this paper, we propose a new hardware-based CEE (CEEv2) that supports virtual memory, while still requiring minimal modifications to the software. Since the virtual memory is managed by the untrusted operating system, the hardware CEEv2 needs a mechanism that detects, or prevents, illegal modifications to the page mappings. For this, we propose a novel mechanism based on an inverted page table. AMD's SEV-SNP [1] employs a Reverse Map Table (RMT) for the same purpose. However, SEV-SNP does not allow a change of mapping once the mapping is validated, whereas the proposed mechanism allows changes of mapping. To determine whether the change is caused by benign swapping, or malicious attacks, we measure and compare the hash of pages being swapped to the disk. The hash computation is not an additional overhead, because it is inevitable in validating the integrity of pages swapped to the disk. In fact, both SEV-SNP and Intel's SGX [2] require hash computation for those pages. The proposed mechanism supports virtual memory and it thwarts virtual-to-physical mapping attacks, while ensuring that any modifications to the software are minimal.

In the following section, we explain the background of verifiable computation and the CEEv2 concept in more detail. After presenting the threat models and the key idea of the proposed mechanism in Section III, we formally define the proposed CEEv2 in Section IV. Section V explains how we implement the CEEv2, followed by experimental results in Section VI. Finally, Section VIII concludes this paper.

II. BACKGROUND AND RELATED WORK

This section describes the concepts of verifiable computation and CEEv2 – i.e., the foundations of the proposed work – and discusses related work in this domain.

A. VERIFIABLE COMPUTATION

When a client (verifier) outsources a computation (F) with an input x to a server (prover), if the verifier wants to verify the correctness of the output y from the prover, the verifiable computation scheme can be employed. According to the verifiable computation scheme, the prover generates a proof (π), by which the verifier can check whether $y = F(x)$ or not.

In the computer system community, attestation techniques have been studied extensively. Since the computation is done by executing instructions stored in memory, the integrity of the instructions in memory is measured and verified [8], [8], [9], [10], [11]. The hash of the instructions is often used as an integrity metric. Even though the instructions are not modified at all, there still exists a possibility of changing the behavior of the computation by control-flow hijacking. To prevent it, the control-flow integrity should also be verified [12], [13], [14], [15]. Even without modifying the instructions and the control flow, it is still possible to distort the computation result by data-only attacks. Data integrity checking has been proposed to prevent such attacks [15], [16], [17], [18]. Even though the integrity is verified by the attestation, there is another issue, known as Time-Of-Check Time-Of-Use (TOCTOU) attack [3]. While the challenge-response protocol works, the prover returns the correct integrity metric, but it may actually execute a different program when the protocol is not active. To address this issue, hardware-software cooperative approaches [19], [20], [21], [22], [23], [24] and employing hardware-based isolation techniques [18], [25], [26], [27], [28], [29] have been proposed.

Attestation techniques pursue a similar goal with verifiable computation, but the latter is more rigorous than the former. While attestation techniques focus on guaranteeing the integrity of computation, verifiable computation also

guarantees that the output is from the given computation and input.

What is guaranteed by the attestation is the integrity of F. Even though F may be correctly executed, a malicious prover may execute F with input x', which is different from the input x given by the verifier. Similarly, a malicious prover may return a different output. The prover may deceive the verifier by executing $y = F(x)$ correctly, but what is actually returned is y'. Without the proof π provided by the verifiable computation, the verifier cannot confirm that the returned output is $F(x)$, and not $F(x')$, nor y'.

More importantly, the security properties guaranteed by verifiable computation are formally proven. *Completeness* guarantees that, if the computation is correct, the proof must be accepted; *soundness* guarantees that, if the computation is *not* correct, there must be negligible probability of accepting the proof. Additionally, verifiable computation schemes may offer *efficiency*, which means minimizing the overhead of checking the proof, and *zero-knowledge*, which means revealing no information other than the computation result.

Verifiable computation has been extensively studied within the context of many applications [5], [6], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50]. In particular, the use of verifiable computation in cloud computing [46], [48], [49], [51] and Internet of Things [47] are active fields of research. Integrity and denial prevention (accurate billing) must be ensured in a cloud-based billing system, and verifiable computation can be applied in a reliable way for verification. The verifiable computation allows the cloud to operate normally and provide accurate results (integrity), and it allows the verifier to prevent denial. There are also studies that employ Verifiable computation for electronic voting [47]. In an electronic voting system, voters need to ensure that their votes are accurately collected and aggregated, while stakeholders need to verify the integrity of the voting system. These crucial requirements motivate the use of verifiable computation to verify the integrity of the election results.

B. CORRECT EXECUTION ENVIRONMENT (CEEv1)

While the verifiable computation schemes guarantee provable security properties, they are extremely slow, because they are implemented with expensive cryptographic algorithms. It takes 10,000 to 100,000 times longer when a computation is performed with proof generation, as compared to the computation without proof generation [7].

To overcome such an excessive overhead, the concept of CEEv1 has been proposed [7]. If we can trust a part of the prover, we do not have to generate a proof for that part, which drastically reduces the overhead. Sealed-Glass Proof [52] is an example of employing Intel's SGX for verifiable computation. The previous work [7] formalizes the requirements of the trusted part and names the qualified part as a CEEv1. Depending on the threat model, a CEEv1 can be implemented by any combination of an operating system, hypervisor, and/or hardware, as long as it meets the formal requirements.

Two requirements are identified for the CEEv1: instruction correctness and state preservation. Instruction correctness means that every instruction must be executed correctly. State preservation means that the state after executing one instruction must remain the same as the state before executing the next instruction. Cryptographic approaches guarantee this by checking the hash of every state in-between instructions [6], [53]. If an operating system, hypervisor, or hardware guarantees this, the verifiable computation scheme does not have to generate a proof for this.

Assuming that the manufacturers of the processor do their best to guarantee instruction correctness, if the trusted part guarantees that the state of the computation cannot be modified by any other software, it is qualified as a CEEv1. Under this definition, Intel SGX is qualified as a hardware CEEv1, as long as OCALL is not allowed. SGX allows the trusted application to call a function of an untrusted application, which is called OCALL. Since the result of OCALL is from an untrusted application, it may distort the state of the protected application. Thus, it should not be allowed for verifiable computation.

This requirement limits the interaction with the world outside of the computation function. It cannot interact with any untrusted software, including the operating system. This limits applicability of the CEEv1. To relax this limitation, a theoretical extension would be required, which is considered beyond the scope of this paper. However, it is still useful for many applications, such as outsourcing heavy computation for digital rights management, data analytics, and machine learning [52].

Compared to a typical Trusted Execution Environment (TEE), CEEv1 has its own pros and cons. CEEv1 does not guarantee confidentiality and disallows any interaction with the world outside of the computation. However, CEEv1 guarantees provable security properties and incurs less performance overhead (4.8% on average [7]).

SGX, SEV-SNP, and CEEv1 all provide protected execution environments using hardware-based mechanisms. This creates a trustworthy computing environment at the hardware level, enhancing resistance to software attacks. Unlike SGX and SEV-SNP, CEEv1 requires minimal changes to the operating system and protected applications. Traditional TEEs require more modifications and cooperation from an untrusted operating system. SGX offers a high level of security but has memory limitations, and certain patterns of memory access can degrade performance. SEV-SNP effectively defends against hypervisor attacks by protecting the entire VM, but it has performance overhead due to encryption/decryption processes. CEEv1 allows for mapping changes and verifies through an Integrity Metric, enabling more flexible memory management. However, it requires trust in the chip vendor, necessitating transparency and strict management across diverse supply chains. Compared to SGX and SEV-SNP, CEEv1 offers advantages such as flexible

memory management, minimal software modifications, performance optimization, and enhanced security. However, challenges remain in managing the hardware trust model and scalability.

C. MOTIVATION FOR THE PROPOSED CEEv2

In a previous work [7], a hardware implementation of a CEEv1 was presented. The hardware implementation generates the hash of the input and the code of the computation (henceforth called a protected application) before starting the application, and the hash of the output when the application completes. The digital signature on those hash values is generated as a proof with the private key of the hardware. To preserve state, the hardware keeps track of the program counter to identify whether it the protected application that is currently running, or if it is any other software. While other software is running, if the target address of a memory instruction is within the address range of the protected application, the memory instruction is blocked. It is formally proven that the hardware meets all the requirements of a CEEv2.

The advantage of this implementation is that it requires minimal modification to the software. It requires no modification to the protected application, and merely a small change to the operating system, which must notify the hardware of the start and end of the protected application. Other parts of the operating system remain unaffected. This advantage could greatly accelerate the adoption of the CEEv2 concept.

While the original incarnation of a CEEv1 environment works well for low-end devices that do not support virtual memory, in most real-world implementations, the computation is more likely to be outsourced to a server that operates using virtual memory. Hence, in this paper, we propose a new hardware-based CEE (CEEv2) implementation that supports virtual memory, while maintaining all the salient advantages of the previous implementation [7].

In summary, the design goals of the proposed hardware-based CEE (CEEv2) are the following:

- It should preserve the state of the protected application (requirement for any CEEv2).
- Its performance overhead should be small (less than 5%).
- It should require minimal modifications to the software.
- Virtual memory should be supported.

III. OVERVIEW

In this section, we define the threat model, explain the main challenges, and present the key idea behind the proposed CEEv2.

A. THREAT MODEL

We trust the hardware and its manufacturer, but not the system software. This is a typical threat model for trusted hardware approaches, even though there are existing techniques that can cope with physical attacks on hardware [54], [55], [56], [57]. Specifically, we assume the following threat model:

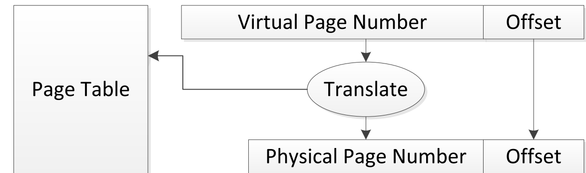

FIGURE 1. The basic virtual-to-physical address translation mechanism.

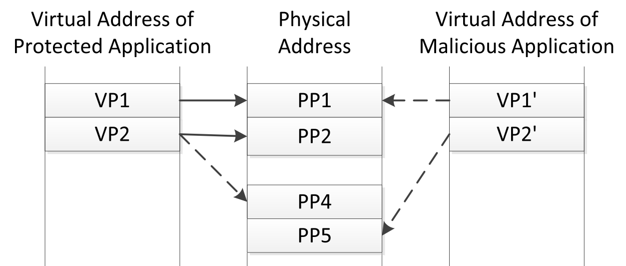

FIGURE 2. An attack scenario of breaking the hardware-only isolation technique if the compromised operating system maliciously counterfeits the virtual-to-physical memory mapping.

- Adversaries can acquire the highest privilege in the system and modify any registers and memory contents, unless they are explicitly protected by the hardware mechanisms.
- Adversaries cannot modify the hardware, nor extract secrets (encryption keys) from the hardware through physical and side-channel attacks.
- The memory controller is integrated with the processor and can control access of the Direct Memory Access (DMA) controller.
- The manufacturer publishes the public key of the hardware and issues a certificate honestly.
- The manufacturer does its best to verify the correctness of the instructions.

We use a secure hash function as a cryptographic primitive. We assume that the secure hash function guarantees unforgeability, which means there is a negligible possibility of two hash values being the same for different input values.

B. SUPPORTING VIRTUAL MEMORY

Since the virtual-to-physical address mapping is managed by the untrusted operating system, it could be exploited by adversaries if the operating system is compromised.

The basic virtual-to-physical address translation mechanism is illustrated in Figure 1. The mechanism is typically implemented in the hardware-based Memory Management Unit (MMU). When a virtual address is given to the MMU, its virtual page number is translated to the physical page number and the offset is concatenated. For the translation, a page table is referred to. The page table is indexed by the virtual page number and the content is the physical page number. The content may also include attributes of the page (e.g., read-only, dirty, etc.). The contents and organization of the page table may differ depending on the MMU implementation.

The page table is stored in main memory and managed by the operating system. The operating system informs the

base address of the table to the MMU. When the context is switched, a different base address is given for a different page table. Since the page table is stored in main memory, it takes time to read it. To reduce access time,a Translation Lookaside Buffer (TLB) is often employed. The TLB is a cache for the page tables. When the context is switched, the TLB should be invalidated to avoid accessing the obsolete page table. Invalidation is usually done by the operating system.

An inverted page table is sometimes used for a special purpose. In our prototype, it is employed to detect changes in the virtual-to-physical mapping. As opposed to the page table, the inverted page table is indexed by the physical page number and the content indicates the virtual page number.

Figure 2 shows example attack scenarios related with the virtual-to-physical memory mapping. Let us suppose that the protected application runs on virtual pages 1 and 2 (VP1 and VP2), which are mapped to physical pages 1 and 2 (PP1 and PP2). When application code is loaded to PP1 and PP2, their integrity metric is measured. If the operating system changes the mapping of VP2 to another physical page (in this example, PP4) after attestation, the protected application does not execute the attested instructions in PP1 and PP2. The compromised operating system may map a virtual page of a malicious application to the physical page used by the protected application. For example (on the right side of Figure 2), if VP1' of a malicious application is mapped to PP1, which is used by the protected application, the malicious application may modify PP1. A more sophisticated attack could be based on the TLB. It is the operating system that is responsible for invalidating the TLB when the context is switched. By delaying invalidation on purpose, the protected application may access a wrong physical page.

C. KEY MOTIVATING IDEA

To prevent those attack scenarios under existing techniques, certain invariants are enforced by hardware [2], or by trusted software [58].

Intel's SGX requires the physical pages assigned to the protected application must be in the Enclave Page Cache (EPC). EPC is in the Processor Reserved Memory (PRM) whose physical address space is fixed at boot time. AMD's SEV-SNP [1] relaxes this constraint by employing an inverted page table, which is called a Reverse Map Table (RMT). Since RMT keeps track of which physical page belongs to which application, it can detect if the compromised operating system maps a virtual page of another application to the physical page of the protected application (the attack scenario illustrated on the right side of Figure 2). To prevent the illegal change of the mapping (the left side of Figure 2), it introduces a new instruction, PVALIDATE. The protected application is supposed to call PVALIDATE only once per virtual page. When it is called, it validates and locks the virtual-to-physical mapping. The mapping cannot be changed thereafter. The physical page may be swapped out to the disk, but it should be swapped in to the same physical page, because the mapping cannot be changed.

Those approaches inevitably limit the capability of the operating system and require modifications to it. Even though the operating system is untrusted, its cooperation is still required. Since the operating system is untrusted, a good amount of complexity is added to prevent its misbehavior [2].

In the proposed CEEv2, this contraint is relaxed further by exploiting the integrity metric. For the physical page belonging to the protected application, its integrity metric needs to be computed when it is swapped out to the disk to prevent illegal modification to it while it is residing in the disk. When it is swapped in, it is now permitted to be allocated to a different physical page, as long as its integrity metric matches.

As for TLB-based attacks, in the proposed technique the memory requests for the integrity metric are issued through the load/store queue in the processor core where the protected application runs. Hence, the memory accesses for the integrity metric are exactly the same as those in the protected application. Furthermore, since the hardware is aware of when the protected application is scheduled in and out, the hardware itself can enforce TLB invalidation.

The operating system maintains its freedom to manage the mapping in the same way the pages of other applications are managed, but the integrity metric will not match if the management is done in an unexpected way. If the operating system changes the mapping to run different code, or maps a physical page of the protected application to a different application, the verifier can detect it by checking the integrity metric. As a result, the proposed technique requires minimal modifications to the operating system, thus enabling easier deployment and smaller attack surface than existing hardware-only approaches.

In addition, we use *context tracking* to identify the current application, instead of using the program counter. As long as the context (registers) is maintained, the hardware assumes that the same application keeps running. When the hardware is informed that the protected application begins, the hardware can keep track of that application by tracking its context.

Intel's SGX provides special instructions to enter and leave the protected application. To check whether the context is correctly restored or not, previous state information is stored in the State Save Area (SSA) [2]. Thus, even if special instructions are provided, context tracking is still required. To minimize modifications to the software, we only employ context tracking to identify whether the protected application is running or not.

IV. THE PROPOSED CEEv2

In this section, we formally define the proposed CEEv2.

A. INPUTS AND OUTPUTS

The proposed hardware-based CEEv2 takes the following inputs from the operating system, which are originally from the verifier.

- F, the code of the protected application. The verifier does not have to send it every time, as it can be stored by the prover.
- x, the input data.
- L, the memory layout of the protected application, which indicates the regions for the code, input, output, and dynamic area (stack and heap).

The CEEv2 generates the following outputs, which are sent to the verifier.

- y, the output of the computation, $y = F(x)$.
- π, the proof of the computation.
  $\pi = Sig(H(F)||H(x)||H(L)||H(y), K_s)$, where Sig is a digital signature scheme, H is the secure hash function, and K_s is the private key of the CEEv2.

After loading the application code (F) and the input (x) to the memory, the operating system notifies the hardware CEEv2 of the start of the protected application with the memory layout information (L). Before starting the protected application, the CEEv2 computes hashes of F, x and L. After completing the protected application, it computes the hash of y and generates π. During the execution, the state is preserved by the mechanism explained in the following subsection.

B. STATE PRESERVATION

For state preservation, the proposed technique maintains the following data structures:

- a, a flag that indicates whether the protected application is running or not.
- $\mathbb{R}$, a set of registers that are backed up when the protected application is scheduled out. $\mathbb{R}$ includes all registers, such as general-purpose registers, program counter, stack pointer, link register, and status register. All registers are compared to $\mathbb{R}$ when the protected application is scheduled in to make sure that the context is restored correctly.
- $\mathbb{D}$: $\mathbb{V} \rightarrow \{0, 1\}$, a set of valid flags. $\mathbb{D}$ is a function where $v \in \mathbb{V}$ must appear only once in the set. $\mathbb{V}$ is a set of virtual pages. When v is accessed for the first time, its flag is turned on, and the pair of v and its mapped physical page is added to $\mathbb{T}$ defined below. $\mathbb{D}$ plays a similar role with PVALIDATE of SEV-SNP [1]. Unlike SEV-SNP, however, the validity is recorded per virtual page (instead of physical page), and it is done automatically by the MMU (instead of introducing a new instruction). The valid flag of v in $\mathbb{D}$ is denoted by $\mathbb{D}[v]$.
- $\mathbb{M}$: $\mathbb{V} \rightarrow \mathbb{H} = \{(v, h) \mid v \in \mathbb{V}\}$, a set of integrity metrics of the swapped pages. $\mathbb{M}$ is a function where $v \in \mathbb{V}$ must appear only once in the set. h is an integrity metric of one page, which is the hash value of page v, i.e. $h = H(v)$, and if v is mapped to p, $h = H(v) = H(p)$. The hash h of a virtual page v in $\mathbb{M}$ is denoted by $h = \mathbb{M}[v]$.
- $\mathbb{T}$: $\mathbb{P} \rightarrow \mathbb{V} = \{(p, v) \mid p \in \mathbb{P}$ and $v \in \mathbb{V}\}$, an inverted page table. $\mathbb{T}$ is a function where $p \in \mathbb{P}$ must appear only once in the set. $\mathbb{P}$ is a set of physical pages. Existence of a mapping indicates its validity. If a particular p cannot be found in $\mathbb{T}$, it means p does not have any valid mapping with a virtual page. For implementation, a valid bit can be used. The virtual page v of a physical page p in $\mathbb{T}$ is denoted by $v = \mathbb{T}[p]$.

In this paper, we explain the state preservation mechanism that protects one application at a time for simplicity. It is straightforward to extend the mechanism for multiple applications by tracking multiple contexts and using a combination of v with its owner application.

a and $\mathbb{R}$ are used for context tracking. The only information needed from the operating system is the start and end of the protected application. Once the protected application gets started, the proposed mechanism keeps track of the running status of the protected application until it ends. Even if the operating system invokes a wrong application intentionally, or accidentally, the verifier can notice it by the proof. $\mathbb{D}$ indicates the validity of the virtual-to-physical address mappings. The reason why $\mathbb{D}$ is needed is explained below. $\mathbb{M}$ is the integrity metric of those pages that have been swapped out to the disk. $\mathbb{T}$ is used to detect a change of mapping. Unlike previous approaches, we allow such changes. An exception is raised only if the integrity metric does not match.

The following two tasks are carried out for context tracking, i.e., maintaining a:

- SwitchOut: While the protected application is running, if any exception occurs, including both hardware and software interrupts, a is turned off, all registers are stored in $\mathbb{R}$, and the TLB is invalidated by hardware.
- SwitchIn: When the context is restored, all registers are compared with $\mathbb{R}$. If they match, a is turned on and the TLB is invalidated by hardware.

The integrity metric is maintained whenever a virtual-to-physical address translation, v to p, occurs. While the protected application is not active (scheduled out), CheckInactive is executed, while CheckActive is executed otherwise. Their pseudocode is given in Algorithm 1.

- CheckInactive: If p is found in $\mathbb{T}$, it means p used to be mapped to $\mathbb{T}[p]$ and now it is swapped to the disk. Thus, its integrity metric is measured and stored in $\mathbb{M}$, if it is not there yet. Then, p is removed from the inverted page table. Note that it could be the operating system that accesses p for page swapping to load a new page. It is also possible that p is assigned to the protected application for a different virtual page v'. In this case, the inverted page table is updated when the protected application accesses p through v'.
- CheckActive: If $\mathbb{D}[v]$ is 0, it means the virtual page is accessed for the first time. In this case, $\mathbb{D}[v]$ is turned on and the inverted page table is updated to the new virtual address, $\mathbb{T}[p] = v$.
  If $\mathbb{D}[v]$ is 1 and $\mathbb{T}[p]$ matches with v, it means the mapping has not changed. Nothing is done for this case.

Algorithm 1 The Pseudo Code for State Preservation

1: **procedure** CheckInactive(v, p)
2:     **if** p is found in $\mathbb{T}$ **then**
3:         Compute $H(p)$
4:         Add ($\mathbb{T}[p]$, $H(p)$) to $\mathbb{M}$
5:         Remove p from $\mathbb{T}$
6:     **end if**
7: **end procedure**
8:
9: **procedure** CheckActive(v, p)
10:     **if** $\mathbb{D}[v]$ is 0 **then**
11:         Update $\mathbb{D}[v] = 1$
12:         Update $\mathbb{T}[p] = v$
13:     **else**
14:         **if** $\mathbb{T}[p]$ is v **then**
15:             Pass
16:         **else**
17:             **if** v is not found in $\mathbb{M}$ **then**
18:                 Search for p' in $\mathbb{T}$ where $\mathbb{T}[p'] = v$
19:                 Abort if p' is not found
20:                 Add (v, $H(p')$) to $\mathbb{M}$
21:                 Remove p' from $\mathbb{T}$
22:             **end if**
23:             Compute $H(p)$
24:             Abort if $H(p)$ does not match with $\mathbb{M}[v]$
25:             Remove v from $\mathbb{M}$
26:             Update $\mathbb{T}[p] = v$
27:         **end if**
28:     **end if**
29: **end procedure**

If $\mathbb{T}[p]$ does not match, it means the mapping has changed. To check its integrity, the integrity metric is looked up in $\mathbb{M}$. If it is not found, it means the mapping has changed, but the previous physical page p' has never been accessed. Thus, the previous physical page must have remained the same. By exhaustive search, the previous physical page is searched for. If it is not found, the application aborts. The hash of the previous physical page p' is added to $\mathbb{M}$ and p' is removed from $\mathbb{T}$. The hash of the new physical page is computed and compared against the previous hash in $\mathbb{M}$. If they match, the application proceeds and the hash is removed from $\mathbb{M}$, because the page is no longer on the disk. The inverted page table is also updated accordingly. If the hash does not match, the application aborts.

If $\mathbb{D}[v] = 1$ and $\mathbb{T}[p] \neq v$ and v is not found $\mathbb{M}$, the previous physical page p' is searched for. If p' is not found, the application aborts, because $\mathbb{D}[v] = 1$ indicates that there must be a previous physical page. In fact, $\mathbb{D}$ is not essential, because it can be determined by the existence of p'. If $\mathbb{T}[p] \neq v$ and v is not found $\mathbb{M}$, and p' is not found, it means this is the first access to v. However, without $\mathbb{D}$, the CEEv2 has to search for p' exhaustively whenever the protected application first accesses a virtual page. To reduce its overhead, we employ $\mathbb{D}$.

C. ILLUSTRATIVE EXAMPLE

Let us take Figure 2 as an example to illustrate how the proposed CEEv2 prevents attack scenarios.

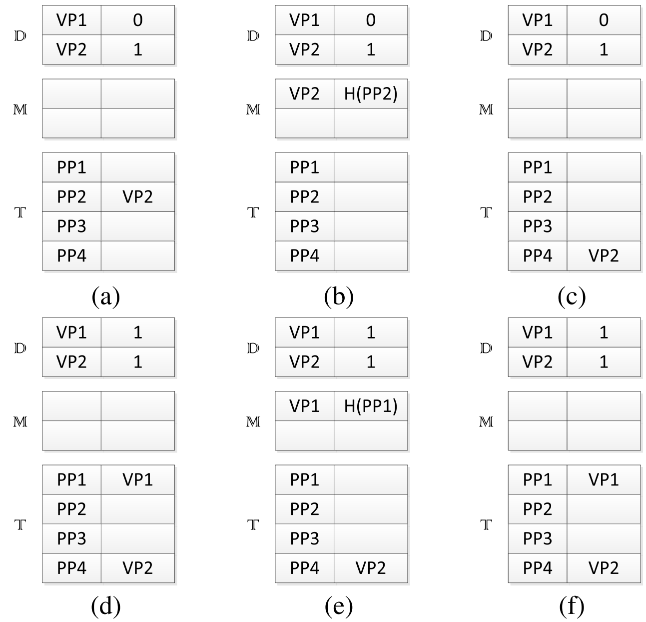

FIGURE 3. Illustration of how the proposed CEEv2 thwarts virtual-to-physical mapping attacks.

When the protected application accesses VP2 for the first time, $\mathbb{D}$[VP2] = 1, and $\mathbb{T}$[PP2] = VP2, as illustrated in Figure 3(a). While the protected application is inactive (scheduled out), let us suppose the compromised operating system changes the mapping of VP2 to PP4. When the proposed application becomes active (scheduled in), if it accesses VP2 again, PP4 is not found in $\mathbb{T}$ and VP2 is not found in $\mathbb{M}$ (Figure 3(a)). Thus, the CEEv2 searches for the previous physical page in $\mathbb{T}$ and finds $\mathbb{T}[PP2]$ = VP2. (VP2, H(PP2)) is added to $\mathbb{M}$, as shown in Figure 3(b). $\mathbb{M}$[VP2] is compared against H(PP4). If the mapping has changed by benign page swapping, H(PP2) must be the same as H(PP4), and the application proceeds as shown in Figure 3(c). If they do not match, it means the compromised operating system tried to make an illegal modification to the state of the protected application by changing the state stored in VP2 from PP2 to PP4.

Let us suppose that VP1' of the malicious application is mapped to PP1. The compromised operating system maps VP1 of the protected application to PP1 as well. When the protected application accesses VP1 for the first time, $\mathbb{D}$[VP1] = 1, and $\mathbb{T}$[PP1] = VP1, as shown in Figure 3(d). While the protected application is inactive, let us suppose the malicious application accesses PP1 through VP1'. Since PP1 is found in $\mathbb{T}$, (VP1, H(PP1)) is added to $\mathbb{M}$. PP1 is removed from $\mathbb{T}$, as illustrated in (Figure 3(e)). When the protected application is active and accesses VP1, PP1 is not found in $\mathbb{T}$, but VP1 is found in $\mathbb{M}$. Thus, H(PP1) is computed and compared against $\mathbb{M}$[VP1]. If the mapping were changed by benign page swapping, the hash must match (Figure 3(f)), or VP1 should have been mapped to another physical page whose hash matches with $\mathbb{M}$[VP1]. If PP1 has been modified

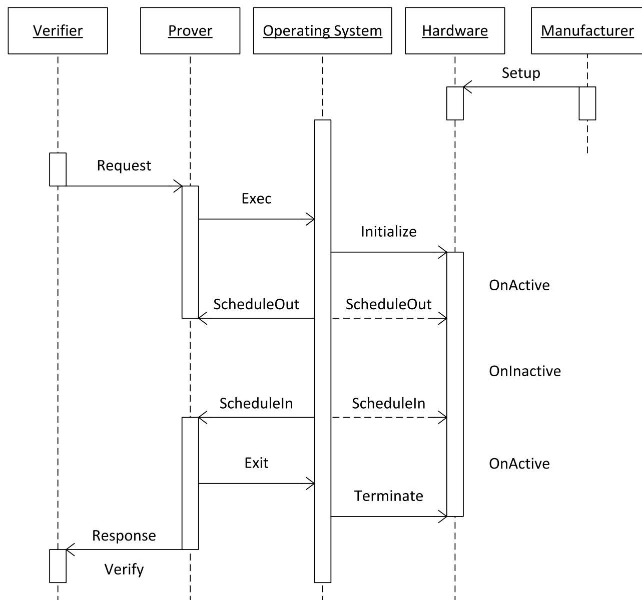

FIGURE 4. **The sequence diagram of the verifiable computation scheme when the proposed CEEv2 is employed.**

by a malicious application, $H(\text{PP1})$ does not match with $\mathbb{M}[\text{VP1}]$, and the application aborts.

D. PROTOCOL

An overview of the verifiable computation scheme employed in the proposed CEEv2 is depicted in Figure 4. The protocol consists of the following algorithms:

- *Setup:* The hardware manufacturer programs a random number to the hardware and the hardware derives a pair of public and private keys from the random number. The hardware keeps the private key (K_s) and publishes the public key (K_p). The manufacturer issues a certificate of the public key.
- *Request:* The verifier sends the program code (F), input data (x), and the memory layout (L) to the prover. The verifier computes and keeps $H(F)$, $H(x)$, and $H(L)$ to verify the response from the prover.
- *Exec:* The prover executes the application with a request of protection to the operating system.
- *Initialize:* The operating system loads F and x to the main memory and informs the hardware of the start of the protected application. It also sends L to the hardware. As soon as the protected application starts, a is turned on and the protection begins.
- *OnActive:* While the protected application is running, CheckActive is executed.
- *ScheduleOut:* If an exception occurs, SwitchOut is executed.
- *OnInactive:* While the protected application is scheduled out, CheckInactive is executed.
- *ScheduleIn:* When the context is restored, SwitchIn is executed.

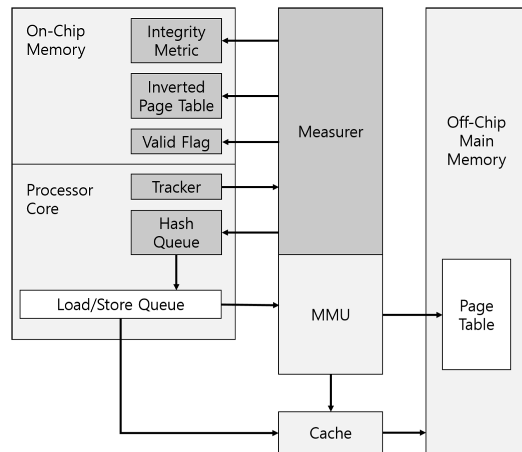

FIGURE 5. **The micro-architecture to implement the proposed technique. The new additional modules required are highlighted in dark grey.**

- *Exit:* When the application finishes, it requests termination of protection.
- *Terminate:* The operating system terminates the protection by the hardware. The hardware returns the output (y) and a proof, which is the concatenation of $H(F)$, $H(x)$, $H(L)$ and $H(y)$, signed by the private key (K_s) of the hardware.
- *Response:* The prover sends the return value (y) of the outsourced application and the proof (π) to the verifier.
- *Verify:* The verifier computes $H(F)||H(x)||H(L)||H(y)$ and verifies the proof with the public key of the CEEv2.

E. LIMITATIONS

The proposed scheme does not allow any interaction with other applications, including the operating system. It does not support file access and I/O, which need to be serviced by the operating system. The only allowed interaction is the primary inputs and outputs from/to the verifier.

File access and I/O can be supported by splitting the protected application. For example, if an application needs to access a file, the application can be split into two: one before the access and another after. The verifier should verify the two applications separately, and forward the output from the first application to the input of the second application. However, if the application needs frequent file accesses and I/O, the performance overhead could be significant.

The proposed scheme does not support confidentiality of the protected application. To support confidentiality, the straightforward way is to encrypt the physical pages of the protected application. However, over the past years, it has been known that memory encryption alone is not enough to guarantee confidentiality. For example, side-channel attacks may reveal the access patterns of the protected application, which may lead to leaking sensitive information [58].

The limitations discussed above stem from the current state of verifiable computation, i.e., they are not supported by existing verifiable computation schemes. If the theoretical framework of verifiable computation is extended to solve

TABLE 1. Comparison of SGX, SEV-SNP, TrustZone, CEEv1 and CEEv2.

Feature	SGX	SEV-SNP	TrustZone	CEEv1	CEEv2
System S/W Modifications	Significant Changes	Moderate Changes	Moderate Changes	Minimal Changes	Minimal Changes
Application Modifications	Significant Changes	Minimal Changes	Significant Changes	Minimal Changes	Minimal Changes
Security Properties	Informal	Informal	Informal	Formal	Formal
Memory Protection	Fixed Memory Allocation	Reverse Map Table	Fixed Memory Allocation	Fixed Memory Allocation	Inverted Page Table
Virtual Memory Support	Yes	Yes	Yes	No	Yes

those issues, the CEEv2 concept can also be extended accordingly.

Table 1 illustrates how SGX, SEV-SNP, TrustZone, CEEv1 and CEEv2 compare across several key features. The proposed CEEv2 supports virtual memory and maintains minimal modifications to the system software, while providing formal security property guarantees that other state-of-the-art designs lack.

This study supports virtual memory, which was not supported by the previously proposed CEEv1, while maintaining the formally proven properties guaranteed by CEEv1. By doing so, CEEv2 ensures the security properties provided by traditional TEEs, while requiring minimal software modifications to use the existing TEE. This is the core of the CEEv2 design.

V. IMPLEMENTATION

The implementation of the proposed CEEv2 requires modifications to the micro-architecture of the processor and minor changes to the operating system. We explain these modifications in this section.

A. MICRO-ARCHITECTURE

The proposed CEEv2 is implemented through the use of new modules added to the MMU and the load/store queue in the processor core. Figure 5 shows the micro-architecture, where the new additional modules required are highlighted in dark grey.

The proposed CEEv2 requires storage for the measurer in the MMU. It stores the valid flags ($\mathbb{D}$), the integrity metric ($\mathbb{M}$), and the inverted page table ($\mathbb{T}$). Since SRAM is known to be safer than external memory (e.g., DRAM), we implement the storage in SRAM. The SRAM can be accessed only by the hardware measurer, and not by the software.

The measurer executes CheckInactive and CheckActive, while a virtual address is being translated by the MMU. It is informed by the tracker in the processor core about the running state of the protected application. When it computes the hash value of a memory page, it puts memory access requests to the hash queue in the load/store queue.

The tracker in the processor core executes SwitchOut and SwitchIn, which keeps track of the running state of the protected application. When context switching occurs, the hardware enforces TLB invalidation.

The SRAM requests for hash computation are put into the hash queue. The load/store queue processes the requests in the hash queue. The requested address given to the hash queue is a virtual address. It makes sure that the address space seen by the measurer is exactly the same as that of the protected application.

The hash computation is triggered by a virtual-to-physical address translation, which is triggered by a memory access. While the hash computation is being performed, the triggering memory access is delayed.

B. DIRECT MEMORY ACCESS CONTROLLER

If a Direct Memory Access (DMA) controller is employed in the system, it may access the physical memory pages without going through the MMU. To handle this, the external memory controller needs to be modified. As Intel's SGX does, we assume that the memory controller is integrated with the processor core and the DMA controller must access the main memory through the memory controller in the processor.

The memory controller prevents any access to the reserved region and interacts with the measurer. It is informed by the measurer about the physical pages used by the protected application. If any attempt is made to access those physical pages, it is reported to the measurer, so that the pair of the affected pages can be removed from the inverted page table.

C. OPERATING SYSTEM

The hardware needs to be informed by the operating system when the protected application starts and ends. Two system calls, which are for launching and terminating an application, are modified. When an application is launched, the operating system notifies the hardware. A compromised operating system may launch a wrong application. However, if a wrong application is launched and executed under protection, the proof cannot match. The verifier can detect it. When the application is terminated, the operating system reads the result and proof from the hardware.

Other than the BIOS and the two system calls, the other parts of the operating system do not need to be modified. Existing algorithms, such as the scheduling algorithm, page management algorithm, and the page-fault handler can be used without modification.

VI. EXPERIMENTAL EVALUATION

In this section, we present experimental results with our prototype implemented on a Field Programmable Gate Array (FPGA). We evaluate the performance overhead of the proposed technique, verify that the proposed technique can

TABLE 2. System parameters of the prototype.

Processor	Type	RISC-V
	Pipeline Stages	5
	Clock Speed	100 MHz
	SRAM	1,288 KB
Main Memory	Type	DDR3
	Size	1 GB
	Clock Speed	200 MHz
Operating System	Type	Linux
	Kernel Version	5.15.4
Cryptography	Secure Hash	SHA-256
	Digital Signature	RSA

thwart attacks of the virtual-to-physical mapping, and analyze the hardware cost.

A. PROTOTYPE

For proof-of-concept, we implemented our scheme by modifying the Rocket chip project, which is based on an open-source RISC-V processor. It was implemented on a Genesys 2 FPGA. The employed operating system is a Linux distribution using Linux kernel version 5.15.4. The system parameters are summarized in Table 2.

The size of SRAM depends on the number of virtual and physical pages that must be supported. The SRAM storage is used to store the integrity metric ($\mathbb{M}$), valid flags ($\mathbb{D}$), and inverted page table ($\mathbb{T}$). The size of $\mathbb{M}$ depends on the size of the virtual memory space to be supported. For example, if we want to support 4 GB (2^{32}) of virtual memory space, the number of necessary virtual pages is 2^{20}, assuming one page has a size of 4 KB (2^{12}). Thus, the number of entries in $\mathbb{M}$ is 2^{20}. The size of each entry is 256 bits (2^5 B), as we employ SHA-256 for hashing. The size of ($\mathbb{M}$) is 2^{25} B (32 MB). The Linux kernel used for experiments supports up to 512 GB of virtual memory space [59]. However, the benchmarks usually do not use all available space. After extensive profiling of the benchmarks used in the presented experiments, we have identified that 2^{13} entries is sufficient for all benchmarks. In this case, the size of $\mathbb{M}$ is 2^{18} B (256 KB).

We need a 1-bit valid flag for each entry of $\mathbb{M}$. Since we have 2^{13} entries, the size of ($\mathbb{D}$) is 1 KB. The size of $\mathbb{T}$ depends on the number of virtual and physical pages. The number of entries is the number of physical pages available in the physical memory. In our prototype, the FPGA board has 1 GB (2^{30}) of main memory. The number of physical pages is 2^{18}, as the page size is 4 KB (2^{12}). An entry of $\mathbb{T}$ stores a virtual page number (13 bits). One more valid bit is also needed. In our prototype, for byte alignment, one entry takes 16 bits (2^2B). The size of $\mathbb{T}$ is 2^{20} B (1 MB). In total, the SRAM size of our prototype is 1,257 KB (256 KB + 1 KB + 1 MB).

For cryptographic computation, we employ SHA-256 for hash and RSA for digital signature. Our proposed technique does not dictate specific hash and digital signature algorithms. We employ these particular ones because they are available from open-source projects [60], [61], but any hash and digital signature algorithms can be used.

TABLE 3. Execution time of benchmarks with and without the proposed scheme.

Benchmark	Original	Proposed	Overhead
600.perlbench_s	27.106s	27.194s	0.32%
605.mcf_s	0.154s	0.154s	0.01%
623.xalancbmk_s	18.687s	18.771s	0.45%
631.deepsjeng_s	843m 4.404s	837m 38.024s	1.19%
641.leela_s	37m 39.755s	37m 38.024s	0.13%
648.exchange2_s	488m 33.202s	503m 8.560s	2.99%
		Average	0.85%

TABLE 4. Execution time of benchmark 600 under protection, with the other benchmarks running concurrently without protection.

Benchmark	Original	Proposed	Overhead
600,605	49s	50s	1.50%
600,605,623	53s	53s	0.86%
600,605,623,631	50s	50s	0.34%
600,605,623,631,641	1m 38s	1m 38s	0.26%
600,605,623,631,641,648	1m 57s	1m 59s	1.86%
		Average	0.86%

The source code of our prototype is open to the public at https://github.com/msms1009/genesys_riscv.git.

B. BENCHMARK APPLICATIONS

We use the SPECspeed Integer benchmarks of the SPEC CPU 2017 suite for evaluation. For the descriptions of the benchmarks, refer to the SPEC website [62].

Due to the limitations of our approach (discussed in Section IV-E), we made the following modifications to the benchmarks. Our approach does not allow the protected application to interact with the kernel. Thus, we load the primary inputs, which are provided by the verifier, to the memory before the application starts, instead of reading them from files. Dynamic memory allocation is replaced by static allocation. Printing messages and writing to files are disabled. Since SPECspeed Integer benchmarks are intended for compute-intensive workloads, they do not frequently interact with the kernel.

In our evaluation, the primary inputs are provided by the verifier. The prover has the executable of the protected application and the verifier has its integrity metric. When the primary inputs are received from the verifier, the prover launches the protected application with the inputs and sends the proof and the output to the verifier.

C. PERFORMANCE EVALUATION

The execution times of the benchmarks with and without the proposed scheme are summarized in Table 3. The benchmarks cover a wide range of execution times, from 1 minute to 853 minutes. The overhead is incurred by interference of memory accesses. The memory accesses of the protected application may be delayed by the access for hash computation. The hash computation itself does not affect the execution time, because it is performed by a

separate hardware module in parallel. Most importantly, the experimental results demonstrate that the overall interference (i.e., the cause of the overhead) is minimal.On average, the performance overhead incurred by the proposed attestation and isolation technique is a near-negligible 0.85%.

The proposed scheme may incur more overhead if page swapping occurs more frequently, because swapping incurs additional hash computation. To test this, we run multiple benchmarks simultaneously. In this experiment, only benchmark 600 is protected, while the other benchmarks run without protection. The execution time of benchmark 600 is compared with the increasing number of unprotected benchmarks. With increasing benchmarks, page swapping is more likely to occur. The results are given in Table 4 and they show that page swapping does not have significant impact on the overhead. This is because, when a page is loaded from the disk to the memory, it is likely to remain in the cache when its hash is computed.

D. SECURITY EVALUATION

We verify that the proposed scheme successfully thwarts the three attack scenarios discussed in Section III pertaining to the virtual-to-physical mapping.

1) ILLEGAL MODIFICATION TO THE MAPPING OF THE PROTECTED APPLICATION

This scenario modifies the protected application's page mapping by modifying the page table such that the virtual page of the protected application is mapped to another physical page.

We added a new system call by modifying the kernel for this scenario. The new system call identifies the process with a pid and uses the `set_pte()` function to map the virtual page of the identified process to another physical page specified by the caller.

The application victim repeatedly prints "Secret Message". Since our prototype does not allow to print messages, we only store the message in a buffer. Subsequently, the attacker application calls the new system function to modify the virtual page mapping that contains "Secret Message" of the victim to the physical page that contains "Hacked Message". When the attack in the corresponding scenario is successfully performed, the victim prints (stores) "Hacked Message" instead of "Secret Message".

This scenario is detected by the second condition of CheckActive, assuming the victim is under the protection of the proposed scheme. The compromised OS modifies the page table of the victim while the victim is scheduled out. When the victim resumes, and if it accesses the virtual page that is now mapped to a different physical page, the virtual page is found in $\mathbb{M}$, but the new mapping cannot be found in $\mathbb{T}$. Thus, the hash of the new physical page is measured again and the new hash does not match the previous one. We confirm that an error is reported to the verifier when this attack scenario is executed.

2) ACCESS ATTEMPT TO PHYSICAL PAGES BEING USED BY PROTECTED APPLICATION

This scenario allows the attacker to modify memory by mapping its virtual page to the physical page of the protected application through an attacker's page table modification.

The attacker modifies its page table using the new system call added in the previous scenario. If the modification is successfully completed, the virtual page of the attacker is mapped to the physical page of the victim, allowing an attacker to modify the memory of the victim.

The scenario assumes the same situation as the previous one. If the attack in the corresponding scenario is successfully performed, the attacker modifies the physical page of the victim and the victim prints the "Hacked Message" repeatedly instead of "Secret Message".

This scenario is detected by the combination of CheckInactive and CheckActive. While the victim is scheduled out, if the attacker accesses the physical page of the victim, CheckInactive finds that the physical address is found in $\mathbb{T}$ and removes the entry from it. When the victim resumes and accesses the virtual page, it cannot be found in $\mathbb{T}$, because it has been removed. Measuring its hash again and finding that it does not match, an error is reported to the verifier.

3) DELAYED INVALIDATION OF TLB

This scenario modifies the applications' page mapping by delaying the TLB invalidation performed during context switching. We implemented the scenario through a modified kernel, two target applications, and the other two applications each notifying the start and end of the attack to the kernel. The kernel is modified by identifying the process name and commenting out the `local_flush_tlb_all()` function executed during context switching between application `attack_target1` and application `attack_target2`. The applications `attack_target1` and `attack_target2` allocate memory to the same virtual address and write the values 0xdeadbeef and 0xcafebabe to the corresponding memory, respectively. This process is repeated within an infinite loop. When the attack in the corresponding scenario is successfully performed, TLB invalidation is delayed when the context switch between `attack_target1` and `attack_target2` happens, resulting in storing an unexpected value to the memory.

This scenario is prevented by the hardware. Recall that the hardware enforces TLB invalidation when the context of the protected application is switched in and out.

E. HARDWARE COST

The hardware cost of our prototype is presented in Table 5. The results are measured by Vivado, the development tool for the FPGA. In total, the number of lookup tables (LUTs) is increased by 153.56%, the registers by 124.36%, and the LUTRAM by 580.42%.

Even though the total overhead is quite high, it is, in fact, largely dominated by the valid flags ($\mathbb{D}$), the integrity metric

TABLE 5. Hardware cost analysis.

Resource	Lookup tables	Registers	LUTRAM
Original	54,453	36,009	4,658
Proposed	138,070	80,788	31,694
Overhead	153.56%	124.36%	580.42%

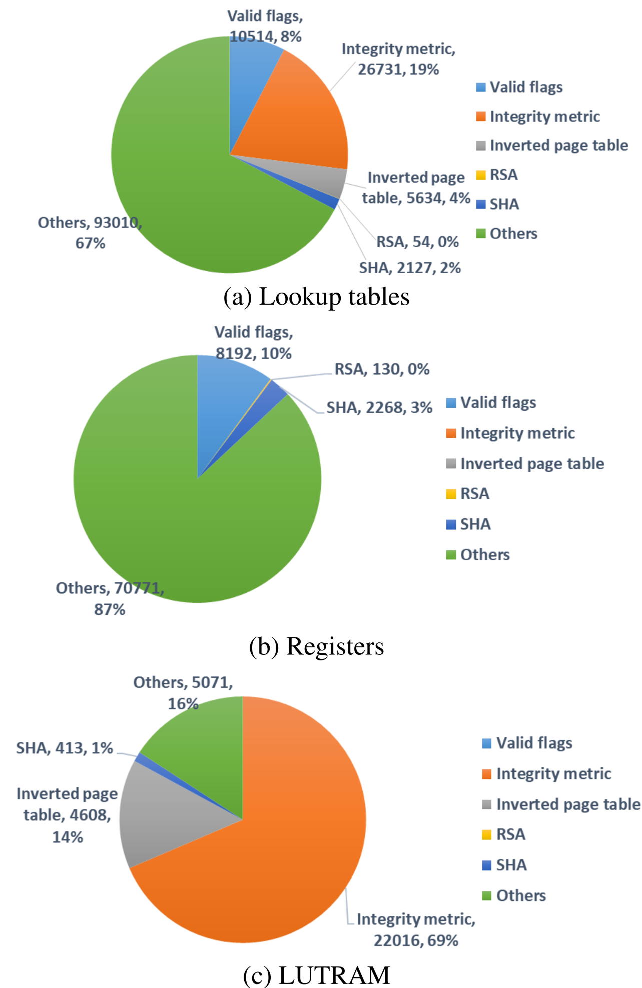

FIGURE 6. Hardware cost breakdown of the proposed architecture.

($\mathbb{M}$), and the inverted page table ($\mathbb{T}$), as shown in the cost breakdown in Figure 6. These three items take up 31.06% of the lookup tables, 10.14% of the registers, and 84.00% of the LUTRAM. Recall that the storage for the measurer (to store the valid flags, the integrity metric, and the inverted page table) was implemented in SRAM for performance. If, instead, it is implemented in DRAM, the performance would degrade, but the cost could be lowered significantly. Specifically, when the storage is implemented in DRAM, the hardware overhead is reduced to 43.72%, 37.94%, and 8.00% for lookup tables, registers, and LUTRAM, respectively.

Furthermore, we employ publicly available SHA and RSA modules for the digital signature. If one employs more optimized hardware implementations, their cost could be reduced as well.

Most importantly, this hardware overhead is over a baseline processor that provides *no security features whatsoever*. Obviously, converting a baseline CPU into an environment that provides extensive security guarantees would – inevitably and justifiably – incur a significant hardware cost. This is the trade-off for providing security guarantees that are mandatory in some domains/sectors.

The proposed CEEv2 has a negligible impact on the overall system overhead except for the hardware cost reported above. One of key benefits of CEEv2 is minimal changes to both the OS and application software. Thus, the impact on the CPU utilization is negligible. The data structures used by CEEv2 could be implemented on the SRAM in the FPGA. The main memory is not affected in terms of bandwidth and space.

VII. DISCUSSION

In this section, we discuss how the proposed CEEv2 in this study can be applied in real situations.

A. POTENTIAL APPLICATIONS

We believe that potential applications for CEEv2 include cloud computing, edge computing, Internet of Things (IoT) devices, and enterprise applications. First, in the case of cloud computing, CEEv2 can be used to ensure the integrity of outsourced computations for cloud service providers. This would be particularly appropriate for processing sensitive data in fields such as healthcare and finance, where data privacy and accuracy are crucial. Second, in the case of edge computing, in scenarios where computing resources are distributed and operate in less secure environments, CEEv2 can provide a robust mechanism to ensure the accuracy of computations performed on edge devices. Third, in the case of IoT devices, IoT often lacks the computational power to perform complex security checks. CEEv2 can offload these tasks to more powerful devices while ensuring the trustworthiness of the results, thereby enhancing the overall security of the IoT system. Lastly, in the case of enterprise applications, within enterprises, CEEv2 can protect computations involving proprietary algorithms and sensitive business data, adhering to data integrity standards and mitigating internal threats.

One of the major challenges is integrating CEEv2 with existing software and hardware systems. Although the proposed CEEv2 requires minimal software modifications, additional effort is needed to ensure compatibility with various and legacy systems. Developing standardized APIs and integration guidelines can facilitate the seamless adoption and integration of CEEv2 with existing systems. CEEv2 relies on trust in chip vendors for its attestation mechanisms. Establishing and maintaining this trust across diverse global supply chains can be challenging. Ensuring hardware transparency and conducting rigorous third-party audits can help mitigate these concerns.

In conclusion, while the proposed CEEv2 offers significant advantages in ensuring the integrity of outsourced computations, several challenges need to be addressed for

successful deployment. By focusing on integration, trust, and scalability, we can promote the widespread adoption and effective utilization of CEEv2.

B. USE CASE

The proposed CEEv2 can be integrated into cloud computing environments as a security layer within hypervisors or container runtimes. For instance, embedding CEEv2 into a Virtual Machine Monitor (VMM) or Docker Engine can provide integrity guarantees for computational tasks executed within virtual machines or containers. Integrating CEEv2 into cloud-based data processing services that handle sensitive customer data ensures that data processing tasks are performed securely, preventing unauthorized modifications or tampering. CEEv2 enhances trust in cloud service providers by assuring customers that their data processing tasks are executed correctly.

For IoT devices, the lightweight implementation of CEEv2 is crucial due to their limited resources. This involves optimizing CEEv2's hardware and software components to fit the constrained computational and memory capabilities of typical IoT devices. In a smart home environment, IoT devices such as thermostats, security cameras, and smart locks process sensitive user data and control critical functions. Integrating CEEv2 into these devices ensures that remote updates or commands are verified and authenticated, providing users with assurance of the integrity and security of their smart home systems.

Exploring these integration strategies highlights the practical applicability of the proposed CEEv2 in enhancing security within cloud computing and IoT device environments.

VIII. CONCLUSION

In this paper, we propose a new CEE (CEEv2) that supports virtual memory. The proposed implementation meets all the pertinent design goals presented in Section II-C. The state of the protected application is preserved by checking for changes in the virtual-to-physical mappings through the inverted page table. Measuring and checking the integrity metric of individual memory pages enables state preservation and allows the existing page-fault handler to be used without any modification. The proposed technique requires modification to only two system calls of the operating system and none to the protected application. The experimental results demonstrate that it incurs a minimal 0.85% performance overhead, while fully supporting virtual memory.

REFERENCES

[1] *AMD SEV-SNP: Strengthening VM Isolation With Integrity Protection and More*, AMD, Santa Clara, CA, USA, 2020.
[2] V. Costan and S. Devadas, "Intel SGX explained," IACR Cryptol. ePrint Arch., Tech. Rep. 2016/086, 2016. [Online]. Available: http://eprint.iacr.org/2016/086
[3] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik, "On the TOCTOU problem in remote attestation," 2020, *arXiv:2005.03873*.
[4] Z. Shan, K. Ren, M. Blanton, and C. Wang, "Practical secure computation outsourcing: A survey," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 1–40, Mar. 2019.
[5] 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.
[6] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von Neumann architecture," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 781–796.
[7] J. Lee, C. Nicopoulos, G. Jeong, J. Kim, and H. Oh, "Practical verifiable computation by using a hardware-based correct execution environment," *IEEE Access*, vol. 8, pp. 216689–216706, 2020.
[8] 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.
[9] X. Carpent, N. Rattanavipanon, and G. Tsudik, "Remote attestation of IoT devices via SMARM: Shuffled measurements against roving malware," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, Apr. 2018, pp. 9–16.
[10] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla, "SCUBA: Secure code update by attestation in sensor networks," in *Proc. 5th ACM Workshop Wireless Secur.* New York, NY, USA: Association for Computing Machinery, Sep. 2006, pp. 85–94.
[11] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of PERipherals' firmware," in *Proc. 18th ACM Conf. Comput. Commun. Secur.* New York, NY, USA: Association for Computing Machinery, Oct. 2011, pp. 3–16.
[12] J. Hu, D. Huo, M. Wang, Y. Wang, Y. Zhang, and Y. Li, "A probability prediction based mutable control-flow attestation scheme on embedded platforms," in *Proc. 18th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun./13th IEEE Int. Conf. Big Data Sci. Eng. (TrustCom/BigDataSE)*, Aug. 2019, pp. 530–537.
[13] N. Koutroumpouchos, C. Ntantogian, S.-A. Menesidou, K. Liang, P. Gouvas, C. Xenakis, and T. Giannetsos, "Secure edge computing with lightweight control-flow property-based attestation," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2019, pp. 84–92.
[14] 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 ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.
[15] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "LiteHAX: Lightweight hardware-assisted attestation of program execution," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2018, pp. 1–8.
[16] 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.
[17] 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.
[18] 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.
[19] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny trust anchor for tiny devices," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.
[20] K. M. E. Defrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and minimal architecture for (establishing dynamic) root of trust," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, vol. 12, 2012, pp. 1–15.
[21] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik, "Temporal consistency of integrity-ensuring computations and applications to embedded systems security," in *Proc. Asia Conf. Comput. Commun. Secur.*, May 2018, pp. 313–327.
[22] A. Ibrahim, A.-R. Sadeghi, and S. Zeitouni, "SeED: Secure non-interactive attestation for embedded devices," in *Proc. 10th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2017, pp. 64–74.
[23] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proc. 9th Eur. Conf. Comput. Syst.*, Apr. 2014, pp. 1–14.
[24] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling, "Sancus 2.0: A low-cost security architecture for IoT devices," *ACM Trans. Privacy Secur.*, vol. 20, no. 3, pp. 1–33, Jul. 2017.

[25] 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.

[26] 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. Depend. Sec. Comput.*, vol. 15, no. 5, pp. 797–810, Sep. 2018.

[27] A. Sensaoui, D. Hely, and O.-E.-K. Aktouf, "Toubkal: A flexible and efficient hardware isolation module for secure lightweight devices," in *Proc. 15th Eur. Dependable Comput. Conf. (EDCC)*, Sep. 2019, pp. 31–38.

[28] H. Dai and K. Chen, "OPTZ: A hardware isolation architecture of multi-tasks based on TrustZone support," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. IEEE Int. Conf. Ubiquitous Comput. Commun. (ISPA/IUCC)*, Dec. 2017, pp. 391–395.

[29] M. Ye, X. Feng, and S. Wei, "HISA: Hardware isolation-based secure architecture for CPU-FPGA embedded systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*. New York, NY, USA: Association for Computing Machinery, Nov. 2018, pp. 1–8.

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

[31] S. Chatel, A. Pyrgelis, J. R. Troncoso-Pastoriza, and J.-P. Hubaux, "Privacy and integrity preserving computations with CRISP," in *Proc. 30th USENIX Secur. Symp. (USENIX Security)*, 2021, pp. 2111–2128.

[32] 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. (USENIX Security)*, 2019, pp. 1429–1446.

[33] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch, B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *Proc. 26th USENIX Secur. Symp. (USENIX Security)*, 2017, pp. 917–934.

[34] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos, "TRUESET: Faster Verifiable set computations," in *Proc. 23rd USENIX Secur. Symp. (USENIX Security)*, 2014, pp. 765–780.

[35] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *Proc. 22nd USENIX Secur. Symp. (USENIX Security)*, 2013, pp. 479–498.

[36] 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. (USENIX Security)*, 2012, pp. 253–268.

[37] L. F. Zhang and H. Wang, "Multi-server verifiable computation of low-degree polynomials," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2022, pp. 596–613.

[38] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Zanella-Beguelin, "EverCrypt: A fast, verified, cross-platform cryptographic provider," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 983–1002.

[39] 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.

[40] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish, "A hybrid architecture for interactive verifiable computation," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 223–237.

[41] 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.

[42] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, and V. Pereira, "A fast and verified software stack for secure function evaluation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 1989–2006.

[43] D. Fiore, C. Fournet, E. Ghosh, M. Kohlweiss, O. Ohrimenko, and B. Parno, "Hash first, argue later: Adaptive verifiable computations on outsourced data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 1304–1316.

[44] M. Backes, D. Fiore, and R. M. Reischuk, "Verifiable delegation of computation on outsourced data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 863–874.

[45] R. Canetti, B. Riva, and G. N. Rothblum, "Practical delegation of computation using multiple servers," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 445–454.

[46] D. Liu, Z. Yan, W. Ding, and M. Atiquzzaman, "A survey on secure data analytics in edge computing," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4946–4967, Jun. 2019.

[47] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[48] J. Ni, K. Zhang, X. Lin, and X. Shen, "Securing fog computing for Internet of Things applications: Challenges and solutions," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 1, pp. 601–628, 1st Quart., 2018.

[49] S. Yi, Z. Qin, and Q. Li, "Security and privacy issues of fog computing: A survey," in *Proc. 10th Int. Conf. Wireless Algorithms Syst. Appl.*, Qufu, China. Cham, Switzerland: Springer, Aug. 2015, pp. 685–695.

[50] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM SIGMOD Rec.*, vol. 34, no. 3, pp. 31–36, Sep. 2005.

[51] R. Gennaro, C. Gentry, and B. Parno, "Non-interactive verifiable computing: Outsourcing computation to untrusted workers," in *Proc. 30th Annu. Cryptol. Conf. Adv. Cryptol. (CRYPTO)*, Santa Barbara, CA, USA. Berlin, Germany: Springer, Aug. 2010, pp. 465–482.

[52] F. Tramèr, 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.

[53] 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.

[54] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "SEDA: Scalable embedded device attestation," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 964–975.

[55] M. Ambrosin, M. Conti, A. Ibrahim, G. Neven, A.-R. Sadeghi, and M. Schunter, "SANA: Secure and scalable aggregate network attestation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 731–742.

[56] A. Ibrahim, A.-R. Sadeghi, G. Tsudik, and S. Zeitouni, "DARPA: Device attestation resilient to physical attacks," in *Proc. 9th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, Jul. 2016, pp. 171–182.

[57] F. Kohnhäuser, N. Büscher, and S. Katzenbeisser, "SALAD: Secure and lightweight attestation of highly dynamic and disruptive networks," in *Proc. Asia Conf. Comput. Commun. Secur.*, May 2018, pp. 329–342.

[58] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp. (USENIX Security)*, 2016, pp. 857–874.

[59] Linux Foundation. *Operating System Interface*. Accessed: Jun. 5, 2023. [Online]. Available: https://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x791.html

[60] N. Iani. (2021). *Sha256 Implementation in Verilog*. [Online]. Available: http://5.9.10.113/67472212/sha256-implementation-in-verilog

[61] R. Singh. (2016). *RSA Cryptosystem Implementation in Verilog*. [Online]. Available: https://github.com/Rajandeep/RSA-CRYPTOSYSTEM-using-verilog

[62] (2017). *SPEC CPU 2017*. [Online]. Available: https://www.spec.org/cpu2017/

DAEHYEON LEE received the B.S. degree from the Department of Cybersecurity, Korea University, Seoul, South Korea, where he is currently pursuing the Ph.D. degree. His current research interests include architecture security and hardware trojan detection. His research interests include computer architecture, microprocessor and computer system design, secure design or hardware-assisted security of processor, non-volatile memory, storage, and hardware trojan detection.

OHSUK SHIN is currently pursuing the integrated M.S./Ph.D. degree with the School of Cyber Security, Korea University, South Korea. His research interests include security of hardware, based on trusted environment.

YEONGHYEON CHA received the B.S. degree from the Department of Cyber Defense, Korea University, Seoul, South Korea. He is currently pursuing the Ph.D. degree with the Department of Cybersecurity, Korea University. His current research interests include privacy preserving technology on cloud infrastructure and confidential computing. His research interests include computer architecture, microprocessor, hardware-assisted security of processor, cloud security, and confidential computing.

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 Georgia Institute of Technology, in 2013. From 2003 to 2008, he was with 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. He has been with the School of Cybersecurity, Korea University, since 2019. His research interests include secure design or hardware-assisted security of processor, non-volatile memory, storage, and dedicated hardware.

TAISIC YUN received the B.S. degree from the Department of Cyber Defense, Korea University, South Korea, in 2022. He is currently pursuing the M.S. degree with the Graduate School of Information Security, KAIST, South Korea. His research interests include system security and software hacking.

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 at Irvine, Irvine, in 2008. She is currently a Professor with the Department of Electrical Engineering, Kookmin University. Her research interests include network security, applied cryptography and fault-tolerant, and distributed computing.

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 Professor with the Department of Information Systems, Hanyang University, Seoul. His research interests include applied cryptography, zero-knowledge proof, and blockchain.

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 was 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.

SANG SU LEE received the B.S. and M.S. degrees in electronic engineering from Kyungpook National University, South Korea, in 1999 and 2001, respectively. Since 2001, he has been with the Electronics and Telecommunications Research Institute, Daejeon, South Korea, where he is currently a Researcher with the Cyber Security Research Division. His recent research interests include cryptography, protocol analysis, and supply-chain cyber security.