SSE: Security Service Engines to Accelerate Enclave Performance in Secure Multicore Processors

Jared Nye and Omer Khan

Abstract—Secure processor technologies leveraging enclaves as their architectural security primitive are frequently deployed in cloud environments. However, enclave-based systems incur performance penalties due to architectural limitations arising from costly enclave exits incurred to interact with system-level software. Exitless calling aims to improve enclave-based performance by spawning additional responder threads alongside the enclave to execute system calls on its behalf, obviating costly enclave exits. However, exitless calling must operate the responder threads as truly asynchronous to the enclave for security isolation guarantees. The self governed timers induce polling stalls that lead to performance loss when enclave and responder threads saturate the available cores in the system. This paper aims to address the polling challenge by introducing security service engines (SSEs) to offload responder threads on dedicated hardware resources. The evaluation shows that SSE equipped secure multicore achieves performance scaling that is at par with a baseline system with no security primitives.

1 Introduction

Cloud computing offers a compelling alternative to costly on-site computing [3], but requires remote clients to rely on third-party cloud providers to process their code and data. This reliance on untrusted third parties raise security concerns addressed in part by the deployment of secure processor technologies [4]. Prevailing secure processor technologies such as Intel Software Guard Extensions (SGX) [4] and AMD Secure Encrypted Virtualization (SEV) [8], as well as upcoming technologies [9], [13], are similar in that they leverage the *enclave* as their essential security primitive. Enclaves provide isolated execution environments protected from both co-resident user- and system-level software, as well as additional security features, such as physical memory confidentiality and integrity, and remote attestation and authentication. Consequently, enclaves are a staple of secure processor technologies and have remained prevalent across both academia [2], [5], [14] and industry [1].

Enclaves offer enhanced security but introduce performance issues. For example, on each enclave memory access, encryption and integrity checking overheads are incurred. Since system-level software may not execute inside enclaves, when an application requires system-levels software services, an enclave exit must be performed which incurs an overhead due to core serialization, state purging, and security checks. To demonstrate how often enclave exits are incurred, we measure the per-core frequency of system-level software interactions for applications commonly used to characterize enclave performance [15], [18], [19]. On an SGX-enabled Intel machine, the average per-core system-level software interactions are observed to be 100K system calls and 50K hypercalls per second. SGX- and SEV-based systems consider the operating system and hypervisor to be untrusted, respectively. This leads to frequent enclave exits for system calls in SGX and hypercalls in SEV, thus making enclave-based processing expensive. Consequently, *exitless* calling was introduced to avoid enclave exits while upholding isolation guarantees between untrusted system-level software and enclaves [15], [19]. Exitless avoids enclave exits through the utilization of an asynchronous worker-responder calling mechanism that spawns two types of threads: (i) *workers* that execute application code inside enclaves, and (ii) *responders* that execute system calls on behalf of enclaves. However, the challenge for exitless calling is that when worker threads saturate the available cores in a system, the additional responder threads incur polling induced stalls due to their asynchronous execution to ensure the isolation property for security [6], [15], [19].

Prior works [16], [17] improve exitless by pinning worker–responder thread pairs on same cores and introduce hardware support for lightweight context switching between them. However, they do not fundamentally address the security-centric polling limitations of exitless. In this paper we make the key observation that if the worker and responder threads execute on dedicated hardware resources, then one can fundamentally address the polling problem in exitless. We propose *Security Service Engines (SSEs)* that offer each enclave core a dedicated hardware context to offload responder threads. In our execution model, each enclave core maps application threads, but at any given time the corresponding SSE engine serves to truly operate in parallel and asynchronously. The lightweight and programmable SSEs provide dedicated hardware in the shared memory paradigm to execute the responder threads. Furthermore, due to their programmability, when exitless calling is not beneficial, the SSE engines can be utilized for other offload services.

The SSEs mitigate the performance bottlenecks of exitless execution for highly parallel cloud applications that effectively utilize the available cores in a multicore. Thus, with responders executing exclusively on SSEs, workers operate uninterrupted and exploit parallelism. Furthermore, polling stalls and context switches are avoided, and costly memory overheads are no longer incurred. To demonstrate the performance benefits with SSEs, we evaluate a web server workload representative of commonly deployed cloud applications on the MIT Graphite multicore simulator [12] modified with the RISC-V instruction set. In comparison to exit-based and exitless enclave execution models, SSEs improve throughput when worker threads match the number of enclave cores in the system. Consequently, the dedicated SSEs for corresponding responder threads deliver at par performance scaling with a baseline system that implements no security primitives.

2 Background and Motivation

Industry and academia have aggressively adopted enclaves to enhance cloud computing security services, using (i) isolation of code/data at the hardware level, (ii) remote attestation and authentication of code/data for remote users, and (iii) encryption of code/data that is only decrypted while being used inside the enclave. Moreover, two types of enclaves have been introduced, *user-level* and *VM-based*, differentiated by the scope of applications they protect. *User-level* enclaves (e.g., Intel SGX) can only encapsulate and execute user-level code. *VM-based* enclaves (e.g., AMD SEV), encapsulate entire virtual machine (VM) instances, and therefore execute code across multiple privilege levels.

Both types of enclaves introduce performance overheads intrinsic to their implementations. For example, to uphold confidentiality and integrity of enclave memory, costly memory encryption and integrity checks are performed on each enclave memory access. Additionally, expensive secure switches outside (and back inside) enclaves are required to enforce isolation from system-level software. These expensive switches are incurred by both user-level and VM-based enclaves, which consider the operating system and hypervisor to be untrusted, respectively. Enclave execution models are described next, followed by their limitations.

• *The authors are with the Department of Electrical and Computer Engineering, University of Connecticut, Storrs, CT 06269 USA. E-mail: {jared.nye, khan}@uconn.edu.*

Manuscript received 17 August 2022; accepted 7 September 2022. Date of publication 27 September 2022; date of current version 10 November 2022.
This work was supported by the National Science Foundation under Grant 1929261.
(Corresponding author: Omer Khan.)
Digital Object Identifier no. 10.1109/LCA.2022.3210149

Exit-based calling implements special procedures for the enclave code to securely access system software services. An untrusted application executes an enclave enter instruction (ecall) that makes a secure context switch into trusted enclave code. Similarly, an enclave exit instruction (ocall) is executed within an enclave to return control back to the untrusted application. During an ecall, hardware security checks are performed, the untrusted state is backed up, the enclave state is restored, and the processor cores are switched into enclave mode. During an ocall, the enclave state is backed up, the untrusted state is restored, the core pipeline is flushed, and the processor cores are switched out of enclave mode. If an ocall is invoked to utilize system-level services, all information located in enclave memory needed by system-level software must first be copied from enclave to untrusted memory. Consequently, entering and exiting enclaves is very costly, and takes 83–113x longer than typical system calls [19].

Exitless calling proposes an alternative model that mitigates expensive enclave exits/enters using an asynchronous calling mechanism between worker threads (executing enclave code) that make system call requests, and responder threads (executing untrusted code) that process system call requests [15], [19]. To uphold isolation guarantees, responders are not allowed to directly access enclave code/data. Therefore, exitless utilizes an asynchronous polling mechanism to create a communication channel between the worker and responder threads using a shared buffer located in untrusted memory. Each responder thread implements a self-governing polling method to periodically check, receive, and process pending system call requests. On the other hand, the corresponding worker thread waits for its pending system call request to complete by monitoring the shared buffer. Once completed, the worker thread copies the result back into trusted memory and continues execution. It has been shown in prior works [6], [19] that exitless calling improves performance by avoiding the exit-based overheads.

Limitations of exitless calling arise from polling and context switching overheads induced by the responder threads. Polling overheads are incurred due to the asynchronous interface between workers and responders and are greatest at increased parallelism. This is because responders continuously poll, even without requests to process. Polling consumes a hardware context without meaningful work until the responder's self-governed timer expires. This results in performance bottleneck when a responder thread is forced to share a core with other worker or responder threads. In such a scenario, a responder polls without meaningful work to process, and reduces the utilization of that core. Moreover, when there are not enough cores for all workers and responders to execute on, the system scheduler is frequently invoked to swap workers for responders and vice versa. This results in costly context switching overheads.

Prior works [16], [17], [18], [19] have introduced thread scheduling and sleeping mechanisms to reduce the time responders poll when few call requests are made. More specifically, [18], [19] reduce responder poll time by having responders set a timer and sleep until it expires. As communication between workers and responders must be truly asynchronous to uphold the isolation property, responders must operate on their own autonomous timers without requiring the workers to invoke them. This requirement makes setting the timer length difficult as responders must sleep often and long enough to reduce polling and context switches, but not so much that wake-up penalties or unnecessary context switching occurs. Therefore, while exitless model allows for improved performance compared to exit-based, it suffers from significant performance challenges when polling becomes the bottleneck.

3 Security Service Engines

We propose to couple each tile (that implements the general-purpose *enclave* core) in a shared-memory multicore with a fully programmable lightweight security service engine (*SSE*) to overcome the polling challenge with exitless mechanism. As shown in Fig. 1,

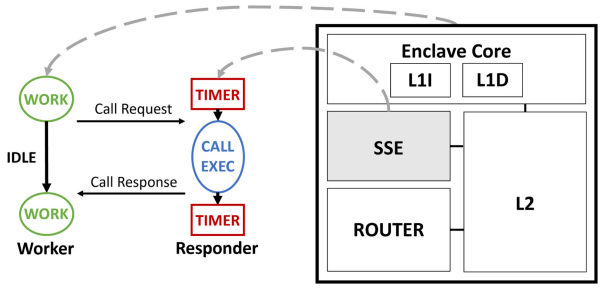

Fig. 1. Tile level view of the proposed architecture with SSE engine integrated alongside general-purpose core and cache hierarchy. The worker and responder threads execute concurrently to exploit multicore parallelism.

SSEs access the shared cache hierarchy via their own dedicated small private L1 instruction and data caches.

The worker-responder architecture is similar to that of exitless, but with two key differences. First, workers execute exclusively on enclave cores, while responders are offloaded to the SSEs. Second, applications are only allowed to spawn as many responder threads as there are SSEs on a machine to avoid unwanted polling bottlenecks. Each SSE operates such that its responder thread strictly serves the worker thread(s) executing on its corresponding enclave core. Thus, while an application may still spawn more worker threads than there are general-purpose enclave cores, any additional performance penalty incurred under SSE because of over-saturating enclave cores will also be incurred by a baseline system without SSEs.

Lightweight Implementation. The work performed by responder threads, i.e., checking the shared buffer and executing system calls, is not compute intensive. Therefore, each SSE utilizes a simple in-order shallow pipeline with no out-of-order execution support. Additionally, during a system call an SSE only performs a one-time copy of data located in untrusted memory. With little data reuse, an SSE implements small and simple private caches. Furthermore, due to the layer of indirection added when workers and responders interact, shared data must be marshalled back-and-forth between private caches of enclave cores and SSEs. For simplicity, an SSE is designed not to cache much of the system call data, but instead evict it to the shared L2 cache. Consequently, the enclave core is able to move the requested data back into its enclave memory without needing to invalidate the SSE private cache, thus avoiding costly sharing misses. These implementation choices result in cost effective caches in SSE without reducing performance.

Benefits. Overall, an SSE enhanced secure multicore provides a novel method to address the polling bottleneck in exitless calling. The responders operate truly asynchronous to the workers, and SSEs can avoid the performance challenges associated with the security-centric polling to service system calls. Consequently, performance gains are achieved by avoiding unwanted thread switches, and reduced system call waiting times for the enclave cores.

Area Overhead. The SSEs come at the cost of additional simple in-order pipeline and minimal private caches. Thus, we consider two multicore implementations: (i) *SSE No Hardware Penalty (SSE-NHP)* that reduces the size of the shared L2 cache in each enclave core to compensate for the hardware overhead of the SSEs, and (ii) *SSE Hardware Penalty (SSE-HP)* that pays for an added cost of SSE in each core with the goal to improve performance.

Generality. Multiple enclave cores may share a single SSE engine to reduce hardware overhead. However, for applications where each worker frequently interacts with system-level software, a one-to-one coupling of enclave cores and SSE engines is necessary to achieve optimal performance. For less interactive applications, a one-to-one coupling of enclave cores and SSE engines is excessive and fewer SSE engines suffice. To optimize resource utilization under such scenarios, the programmable SSE engines can be reused for other useful services such as cache, prefetching, or even power management optimizations.

4 Methodology

The baseline (BL), exit-based (EX-B), exitless (EX-L), and our proposed SSE-enhanced architecture (SSE) are implemented using the MIT Graphite simulator enhanced with the needed performance models, and RISC-V ISA [7], [12]. A 64-core tiled shared memory multicore processor is modeled with a two-level coherent private $32KB$ L1-I and L1-D, shared $256KB$ L2 cache hierarchy per core, resulting in a $16MB$ total shared cache capacity. Each tile's enclave core implements an out-of-order RISC-V pipeline. The tiles are interconnected using a 2D mesh on-chip network with X-Y routing.

4.1 Enclave Modeling

To model the performance overheads of memory encryption and integrity checking, a constant latency of 10 cycles is added to each main memory data access performed across each implementation excluding BL.

Exit-based (EX-B). Tian et al. [17] quantify the overhead of each ecall and ocall in Intel SGX to be 9,000 cycles, respectively. Thus, a 9,000-cycle latency is added to each ecall and ocall performed in EX-B.

Exitless (EX-L). When the number of threads exceed the core count (at 64 workers), each worker-responder pair is pinned to a single core as this yields the highest performance under such conditions. As workers and responders cannot execute simultaneously on the same core, they are swapped out for one another based on a timer set by the responder. Initially, responders check the shared buffer for requests, and if there are none, set a timer and sleep until it expires. After a responder goes to sleep, a worker is swapped in and executes until the timer set by a responder expires.

The cost of a context switch is set to a conservative $500ns$ based on measurements performed in [10]. Responder sleep times are configured to $10\mu s$ which was experimentally determined to achieve optimal performance for the evaluated workload.

Security Service Engines (SSE). SSEs are modeled by adding an additional in-order simple pipeline and its own private $1KB$ L1 caches to each of the 64 tiles. Each SSE placed in a tile shares the L2 cache slice with the corresponding enclave core. Two variants of SSEs are evaluated: (i) *SSE-NHP* which has a reduced $224KB$ L2 slice on each tile to compensate for the SSE area overhead and ensure resources comparable to BL, EX-B, and EX-L are used, and (ii) *SSE-HP*that has a full $256KB$ L2 slice on each tile. Threads are spawned such that workers are mapped to enclave cores, while responders are mapped to the SSEs.

4.2 Workload

To systemically study enclave performance at increased parallelism, we utilize a workload that is able to execute on an application-level simulator as porting real applications to a simulator is not feasible. We develop a representative workload, *Server*, that is highly interactive and modeled to match the steady-state behavior of the web-server lighttpd [11] used by several highly cited works on SGX [15], [18], [19].

Server spawns N workers that serve M/N web page requests that are divided evenly between workers, where M is the total number of web page requests issued. After accepting a connection request, workers iterate over a loop that begins with *ioctl* call to get the request size, then allocates a buffer of that size. The request is read using *recv*call, and parsed which also gets the requested file name, then the file is opened using *open*call. *Fstat* is then called to get the needed file stats needed that generates the response sent to the client using *send* call. *Fstat* is called again to ensure the file has not been modified, then *lseek* is called to move the file offset to the beginning of the file. A buffer is allocated to store the file contents that are read and then sent using *read* and *send* calls, respectively. Finally, the worker closes the file using *close* call, then repeats this process for the next request.

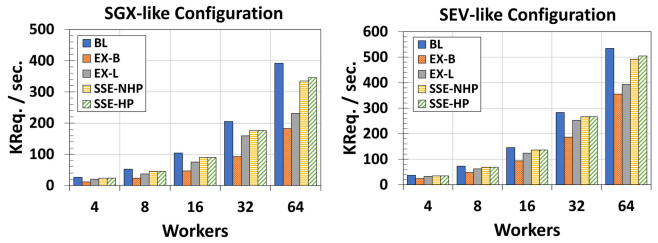

Fig. 2. Server throughput performance.

Server is evaluated under two configurations: (i) an *SGX-like configuration* that incurs an enclave exit upon each traditional system call mentioned in the previous paragraph, and (ii) an *SEV-like configuration* that only incurs enclave exits upon *send* and *recv* system calls as these communicate directly with devices that are managed by the hypervisor. 1000 requests, each with web page size of $16KB$ are issued evenly between clients, with each worker serving its respective client's requests. Throughput is measured in thousands of requests served per second and all evaluations are performed with 4 to 64 worker threads.

4.3 Metrics

The performance of each implementation is measured by tracking the parallel completion time across all workers. To gain further insights, latency measurements are also tracked as follows:

- *Workload* is the time spent executing workload-specific tasks that are not system calls or hypercalls.
- *Pack* and *Unpack* are the time spent copying data from enclave memory to untrusted memory and the time spent copying data to enclave memory from untrusted memory, respectively.
- *Syscall* is the time spent processing system calls and hypercalls for the SGX-like and SEV-like configurations, respectively.
- *Polling* is the time (i) workers wait for a responder's timer to expire after making a request and (ii) responders spend checking the shared buffer for requests.
- *Ecall* and *Ocall* are the time spent performing ecalls and ocalls, respectively.
- *Ctx Switch* is the time spent executing a context switch, which is incurred on each swap of workers and responders in EX-L.

5 Evaluation

BL achieves the highest throughput across both workload configurations as the number of workers are increased from 4 to 64 (Fig. 2). EX-B is able to achieve performance scaling under both configurations but does consistently worse due to costly ecalls/ocalls. EX-L meanwhile achieves performance scaling relatively in line with the baseline up to 32 workers for both configurations, but suffers a decrease in performance at 64 workers, primarily due to polling overheads. The context switching overheads also contribute to the performance degradation observed in EX-L.

SSE behaves identically to EX-L up to 32 workers, but continues to scale at 64 workers in both configurations as the additional service engines enable SSE to mitigate polling and context switching overheads. As we are focused on improving EX-L performance specifically when the number of threads exceed the number of cores, we show the normalized breakdown of 64-worker configurations in Fig. 3, and use this as the basis of our analysis for each implementation's performance behavior.

5.1 Exit-Based (EX-B) and Exitless (EX-L) Analysis

Compared to BL, EX-B incurs a massive ecall/ocall overhead on each system call. These ecall/ocall overheads lead to a 2.2× and 1.5× performance loss for the SGX-like and SEV-like configurations, respectively.

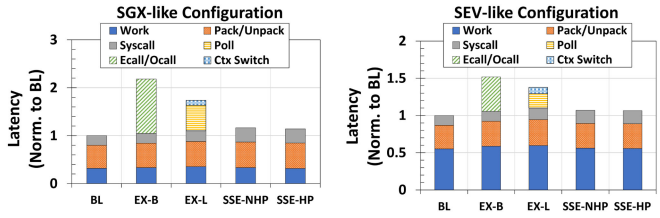

Fig. 3. 64-worker latency breakdown (normalized to BL).

At 64-workers, EX-L performance is 1.7× and 1.4× worse than BL for SGX-like and SEV-like configurations, respectively. This happens due to the frequent swapping of workers and responders as there are more threads than cores. Frequent context switching leads to (i) cache thrashing since data lost during a context switch must be re-fetched, and (ii) increased polling time as responders check for system call requests on each invocation. Fig. 3 shows polling overheads are the primarily cause of performance degradation in EX-L. Even with an ideal 0-cycle context switching overhead, EX-L would still trail BL performance by 1.6× and 1.3× for the SGX-like and SEV-like configurations, respectively. This demonstrates any EX-L configuration that relies on the security-centric timers will fail to match BL.

L1D cache thrashing is exacerbated by polling in EX-L compared to BL. In the SGX-like configuration an increase in L1D and L2 cache misses of 29% and 23%, respectively, are incurred compared to BL which contribute to a 10% and 11% increase in workload and system call time, respectively. For the SEV-like configuration, cache thrashing leads to an increase in L1D and L2 cache misses of 88% and 29%, respectively, compared to BL which further causes an 8.2% and 18.7% increase in workload and system call time, respectively.

5.2 Security Service Engine (SSE) Analysis

SSE eliminates ecall/ocall and polling overheads incurred in both EX-B and EX-L (Fig. 3). However, the worker–responder interactions that are not present in BL are now split across enclave cores and SSEs, adding undesirable data movement between them that prevents SSE from matching BL performance.

Both SSE variants with and without hardware overhead penalty incur a greater memory access latency than BL due to added memory system stress caused by data movement in worker-responder interactions. Consequently, compared to BL an increase in L1D sharing misses of 4.8× and 2.4× are incurred by SSE-NHP and an increase in L1D sharing misses of 4.4× and 1.4× are incurred by SSE-HP for the SGX-like and SEV-like configurations, respectively. Compared to BL, SSE-NHP (which has a reduced L2 cache per tile), incurs 16.1% and 11.8% more L2 misses, while SSE-HP incurs 0.2% and 0.1% more L2 misses for the SGX-like and SEV-like configurations, respectively. These increases in memory access latencies result in a performance loss in the workload, pack, unpack, and system call processing components of execution latency. Furthermore, SSE-NHP has longer work and system call processing times than SSE-HP from increased L2 misses.

Overall, our evaluation indicates that as the total number of threads exceed the core count of a machine, exitless calling becomes less beneficial because of significant polling and context switching overheads. However, by providing enough service engines such that the hardware's parallelism is not overburdened as proposed in the SSE architecture, polling and context switching overheads can be obviated and continued performance scaling is achieved.

6 Conclusion and Future Work

This paper proposes a novel heterogeneous secure multicore architecture that leverages dedicated service engines to improve the performance scaling of applications executing inside enclaves. The preliminary evaluations show unnecessary stalls and frequent context switches present in enclave exitless calling can be obviated and performance scaling can be achieved even when the number of workload threads exceed the number of cores. Our future work will incorporate the following enhancements for the proposed SSE architecture:

1) Reducing the system call processing overheads in SSE is achievable by leveraging asynchronous execution of SSEs and enclave cores. Enclave cores may continue to perform useful work rather than busy poll after making a system call request. We will explore a novel asynchronous calling API that will allow workers to perform useful work while responders process system call requests in parallel.
2) We will evaluate a diverse range of workloads to demonstrate the applicability of the proposed SSE architecture. Workloads will be developed from the domains of web servers, database systems, encryption services, graph processing, and machine learning on images and graphs. We will provide detailed analysis of the microarchitecture effects across all workloads, and also study various area or performance efficient implementations of SSEs.

References

[1] R. Boivie and P. Williams, "SecureBlue++: CPU support for secure executables," IBM, Yorktown Heights, NY, USA, Tech. Rep., 2013.
[2] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure enclaves in a speculative out-of-order processor," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 42–56.
[3] D. C. Chou, "Cloud computing: A value creation model," *Comput. Standards Interfaces*, vol. 38, pp. 72–77, 2015.
[4] V. Costan and S. Devadas, "Intel SGX explained," Comput. Sci. Artif. Intell. Lab. Massachusetts Inst. Technol., Cambridge, MA, Tech. Rep. 086, 2016.
[5] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Conf. Secur. Symp.*, 2016, pp. 857–874.
[6] B. D'Agostino and O. Khan, "Seeds of SEED: Characterizing enclave-level parallelism in secure multicore processors," in *Proc. Int. Symp. Secu. Private Execution Environ. Des.*, 2021, pp. 203–209.
[7] H. Dogan, M. Ahmad, B. Kahne, and O. Khan, "Accelerating synchronization using moving compute to data model at 1,000-core multicore scale," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 1, pp. 4:1–4:27, Feb. 2019.
[8] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White paper, Apr. 2016.
[9] D. Lee, D. Kohlbrenner, S. Shinde, D. Song, and K. Asanović, "Keystone: An open framework for architecting TEEs," 2019, *arXiv: 1907.10119*.
[10] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop Exp. Comput. Sci.*, 2007, Art. no. 2–es.
[11] Lighttpd, "An open-source web server optimized for speed-critical environments." [Online]. Available: https://www.lighttpd.net/
[12] J. E. Miller et al., "Graphite: A distributed parallel simulator for multicores," in *Proc. 16th Int. Symp. High-Perform. Comput. Architecture*, 2010, pp. 1–12.
[13] D. P. Mulligan, G. Petri, N. Spinale, G. Stockwell, and H. J. M. Vincent, "Confidential computing—a brave new world," in *Proc. Int. Symp. Secure Private Execution Environ. Des.*, 2021, pp. 132–138.
[14] H. Omar and O. Khan, "IRONHIDE: A secure multicore that efficiently mitigates microarchitecture state attacks for interactive applications," in *Proc. IEEE Int. Symp. High Perform. Comput. Architecture*, 2020, pp. 111–122.
[15] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS services for SGX enclaves," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 238–253.
[16] O. Shafi and J. Bashir, "SecSched: Flexible scheduling in secure processors," in *Proc. ACM Int. Conf. Parallel Architectures Compilation Techn.*, 2020, pp. 229–240.
[17] H. Tian et al., "Switchless calls made practical in intel SGX," in *Proc. 3rd Workshop System Softw. Trusted Execution*, 2018, pp. 22–27.
[18] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2017, pp. 645–658.
[19] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves," in *Proc. 44th Annu. Int. Symp. Comput. Architecture*, 2017, pp. 81–93.