IceClave: A Trusted Execution Environment for In-Storage Computing

Luyi Kang*†
University of Maryland, College Park

Yuqi Xue*
UIUC

Weiwei Jia*
UIUC

Xiaohao Wang‡
UIUC

Jongryool Kim
SK Hynix

Changhwan Youn
SK Hynix

Myeong Joon Kang
SK Hynix

Hyung Jin Lim
SK Hynix

Bruce Jacob
University of Maryland, College Park

Jian Huang
UIUC

ABSTRACT

In-storage computing with modern solid-state drives (SSDs) enables developers to offload programs from the host to the SSD. It has been proven to be an effective approach to alleviate the I/O bottleneck. To facilitate in-storage computing, many frameworks have been proposed. However, few of them treat the in-storage security as the first citizen. Specifically, since modern SSD controllers do not have a trusted execution environment, an offloaded (malicious) program could steal, modify, and even destroy the data stored in the SSD.

In this paper, we first investigate the attacks that could be conducted by offloaded in-storage programs. To defend against these attacks, we build a lightweight trusted execution environment, named IceClave for in-storage computing. IceClave enables security isolation between in-storage programs and flash management functions that include flash address translation, data access control, and garbage collection, with TrustZone extensions. IceClave also achieves security isolation between in-storage programs by enforcing memory integrity verification of in-storage DRAM with low overhead. To protect data loaded from flash chips, IceClave develops a lightweight data encryption/decryption mechanism in flash controllers. We develop IceClave with a full system simulator. We evaluate IceClave with a variety of data-intensive applications such as databases. Compared to state-of-the-art in-storage computing approaches, IceClave introduces only 7.6% performance overhead, while enforcing security isolation in the SSD controller with minimal hardware cost. IceClave still keeps the performance benefit of in-storage computing by delivering up to 2.31× better performance than the conventional host-based trusted computing approach.

*Co-primary authors.
†Work done while visiting the Systems Platform Research Group at UIUC.
‡Now at NVIDIA.

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480109

CCS CONCEPTS

• **Security and privacy** → **Hardware security implementation**; • **Computer systems organization** → **Architectures**; • **Hardware** → **External storage**.

KEYWORDS

In-Storage Computing, Trusted Execution Environment, Security Isolation, ARM TrustZone

ACM Reference Format:
Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce Jacob, and Jian Huang. 2021. IceClave: A Trusted Execution Environment for In-Storage Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480109

1 INTRODUCTION

In-storage computing has been a promising technique for accelerating data-intensive applications, especially for large-scale data processing and analytics [15, 20, 33, 39, 45, 52, 56, 57, 72, 81]. It moves computation closer to the data stored in the storage devices like flash-based solid-state drives (SSDs), such that it can overcome the I/O bottleneck by significantly reducing the amount of data transferred between the host machine and storage devices. As modern SSDs are employing multiple general-purpose embedded processors and large DRAM in their controllers, it becomes feasible to enable in-storage computing in reality today.

To facilitate the wide adoption of in-storage computing, a variety of frameworks have been proposed. For instance, Willow [72] enabled developers to offload code from the host machine to the SSD via RPC protocols, and Biscuit [39] developed an in-storage runtime system for supporting multiple in-storage computing tasks following the MapReduce computing model. All these prior works show the great potential of in-storage computing for accelerating data processing in data centers. However, most of them [19, 33, 34, 52, 56, 57, 72, 82] focus on the performance and programmability, few of them treat the security as the first citizen in their design and implementation, which imposes great threat to the user data and SSD devices, and further hinders its widespread adoption.

As in-storage processors operate independently from the host machine, and modern SSD controller does not provide a trusted execution environment (TEE) for programs running inside the SSD, they pose severe security threats to user data and flash chips. To be specific, a piece of offloaded (malicious) code could (1) manipulate the mapping table in the flash translation layer (FTL) to mangle the data management of flash chips, (2) access and destroy data belonging to other applications, and (3) steal and modify the memory of co-located in-storage programs at runtime.

To overcome these security challenges, as developed in state-of-the-art in-storage computing frameworks [39, 72], we can simplify the runtime system by maintaining a copy of the privilege information in the DRAM of SSD controllers (SSD DRAM) and enforcing permission checks for in-storage programs. However, such a solution still suffers from many security vulnerabilities. For example, a malicious offloaded program can exploit memory vulnerabilities such as buffer overflow [28, 79, 88] to enable privilege escalation to access and modify the cached mapping table of FTL in the SSD DRAM; adversaries can steal and modify intermediate data and results generated by the co-located in-storage programs via physical attacks such as cold-boot attack, bus snooping attack, and replay attack [67, 71, 86]. An alternative approach is to adopt Intel's Software Guard Extension (SGX) as a drop-in solution. Unfortunately, modern in-storage processor architectures do not support SGX techniques. And the SGX approach still suffers from significant performance overhead [11, 25, 50, 63, 70, 80], which cannot be afforded by in-storage computing and SSD controllers today.

Therefore, providing a secure, lightweight, and trusted execution environment for in-storage computing is an essential step towards its widespread adoption. Ideally, we wish to enjoy the performance benefits of in-storage computing, while enforcing the security isolation between in-storage programs, the core FTL functions, and physical flash chips, as demonstrated in Figure 1.

To this end, we present IceClave, a trusted execution environment for in-storage computing. Unlike generic TEE solutions such as SGX [25], IceClave is designed specifically for modern SSD controllers and in-storage programs, with considering the unique flash properties and in-storage workload characteristic. With ensuring the security isolation, IceClave includes (1) a new memory protection scheme to reduce the context switch overhead incurred by flash address translations; (2) an optimization technique for securing in-storage DRAM for in-storage programs by taking advantage of the fact that most in-storage applications are read intensive; (3) a stream cipher engine for securing data transfers between storage processors and flash chips, with low performance overhead and energy consumption; and (4) a runtime system for managing the life cycle of in-storage TEEs.

Specifically, to achieve the security isolation between the in-storage program and the FTL (❶ in Figure 1), we extend the TrustZone of ARM processors available in a majority of SSD controllers. IceClave executes core FTL functions such as garbage collection and wear leveling in the secure world, and runs the offloaded programs in the normal world, such that offloaded programs cannot intervene the flash management. To protect the mapping table of FTL with low overhead, we introduce a protected memory region in the normal world, and place the address mapping table in it to avoid

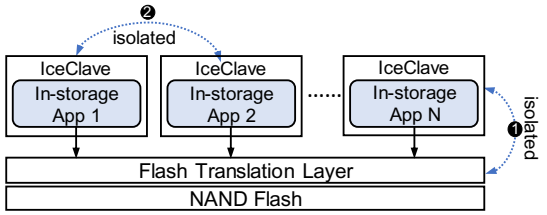


Figure 1: IceClave enables in-storage TEEs to achieve security isolation between in-storage programs, FTL, and flash chips. The shaded components are untrusted.

context switches for flash address translation. Therefore, only protected FTL functions can update the mapping table, and offloaded in-storage programs can only read it for address translation with enforced permission checks.

In order to achieve the security isolation between in-storage programs (❷ in Figure 1), we build in-storage TEEs to host the offloaded programs, and enforce data encryption and memory integrity checks in both data communication and processing for in-storage programs. Since most in-storage computing workloads are read intensive, IceClave mainly needs to conduct the integrity check for the intermediate data and results generated by in-storage programs, which does not introduce much performance overhead to the in-storage program execution. To secure flash pages read by in-storage programs running in the in-storage TEE, we also integrate a lightweight stream cipher engine into the SSD controller with minimum hardware cost. In summary, we make the following contributions in this paper:

- We present a trusted execution environment for in-storage computing, in which it protects core FTL functions from malicious in-storage programs with TrustZone extension.
- We support security isolation between in-storage programs by enabling lightweight memory encryption and integrity verification for SSD DRAM.
- We show the required hardware and software extensions for IceClave are minimal and feasible to be developed in modern SSD controllers.
- We develop a system prototype using an SSD FPGA board for a quantitative overhead study of IceClave, and a full system simulator for sensitivity analysis.

Specifically, we implement IceClave with a full system simulator gem5 [37], and integrate an SSD simulator SimpleSSD [38] and memory simulator USIMM [21] into it for supporting secure in-storage computing. We also develop a system prototype to verify the core functions of IceClave with a real-world OpenSSD Cosmos+ FPGA board [84]. We use a variety of data-intensive applications that include transactional databases to evaluate the efficiency of IceClave. Compared to state-of-the-art in-storage computing approaches, IceClave introduces only 7.6% performance overhead to the in-storage runtime, while adding minimal area and energy overhead to the SSD controller. Our evaluation also demonstrates that IceClave can maintain the performance benefit of in-storage computing by delivering 2.31× better performance on average than the conventional host-based computing approach.

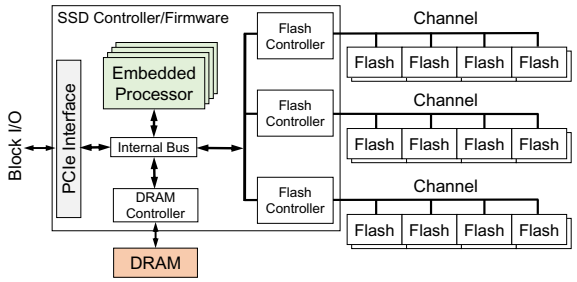


Figure 2: Internal architecture of flash-based SSDs.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce the background of flash-based SSDs and in-storage computing, and discuss why it is desirable to have TEEs for in-storage computing.

2.1 Flash-Based Solid-State Drive

The rapidly shrinking process technology has allowed SSDs to boost their performance and capacity, which accelerates their adoption in commodity systems such as data centers today [4, 43, 44]. We present a typical SSD architecture in Figure 2. An SSD has three major components: a set of flash memory packages, an SSD controller having embedded processors with DRAM, and flash controllers [23, 56]. The flash packages are organized in a hierarchical manner. Each SSD has multiple channels. Each channel is shared by multiple flash packages. Each package consists of multiple flash chips. Each chip has multiple planes. Each plane is divided into multiple flash blocks, each of which consists of multiple flash pages.

Due to the nature of flash memory, when a free flash page is written once, that page is no longer available for future writes until that page is erased. However, erase operation can be performed only at a block granularity, which is time-consuming. Thus, writes are issued to free pages that have been erased (i.e., out-of-place write) rather than waiting for the expensive erase operation. Garbage collection (GC) will be performed later to clean the obsolete data in SSDs. As each flash block has limited endurance, it is important for blocks to age uniformly (i.e., wear leveling). SSDs employ out-of-place write, GC, and wear leveling to overcome the shortcomings of SSDs and maintain indirections for indexing the logical-to-physical address mapping. All these are managed by the Flash Translation Layer (FTL) in the SSD controller.

2.2 In-Storage Computing

We have long recognized the inefficiency of traditional CPU-centric computing for data-intensive applications that need to transfer large amounts of data from storage. The application performance is limited by the low bandwidth of PCIe interface between the host and SSD. To tackle this challenge, various in-storage computing approaches [39, 52, 56, 72, 89] have been proposed. With them, we can process data by exploiting SSD processors and high internal bandwidth of flash chips. Their significant performance benefits show that in-storage computing is a promising technique.

However, as in-storage processors operate independently from the host, it poses security challenges to the adoption of in-storage computing, especially in the multi-tenancy setting where multiple application instances share the physical SSD [49, 55, 74–76, 92].

2.3 In-Storage Vulnerabilities

When specific code is offloaded to in-storage processors, a copy of the privilege information is transferred and maintained in the DRAM of the SSD controller [39, 72]. Such a solution is developed under the assumption that the offloaded code has already known the address and size of the accessed data in advance. However, adversaries can exploit in-storage software and firmware vulnerabilities such as buffer overflows [28, 79, 88], and bus-snooping attack [71] to achieve privilege escalation. After that, they can conduct various further attacks. We present them in the following.

- A malicious user can manipulate the intermediate data and output generated by in-storage programs via both software and physical attacks, causing incorrect computing results.
- A malicious program can intercept FTL functions like GC and wear leveling in the SSD and mangle the flash management. This would cause data loss or device destroyed.
- A malicious user can steal user data stored in flash chips via physical attacks like bus snooping attack, when in-storage programs load data from flash chips to SSD DRAM.

To defend against these attacks, an alternative solution is to develop an OS or hypervisor for in-storage computing. However, due to the limited resources in the SSD controller, running a full-fledged OS can introduce significant overheads to the SSD and increase the attack surface, due to its large codebase. Moreover, these techniques are not sufficient to defend against the aforementioned attacks such as board-level physical attacks. As we move compute closer to storage devices, it is highly desirable to have a lightweight execution environment for this non-traditional computing paradigm.

Since SSDs were designed with the assumption they are hardware isolated from the host and purely used as storage rather than computing, modern computing systems do not provide secure runtime environment for in-storage computing. As discussed in §1, a straightforward approach is to adopt the SGX-like solutions [25, 54]. However, this requires significant hardware changes and even replacement of storage processors available in modern SSDs. As SGX was developed as a generic framework for host machines, it is hard to achieve optimal performance for in-storage programs.

3 THREAT MODEL

In this work, we target the multitenancy where multiple application instances operate in the shared SSD. Following the threat models for cloud computing today [1, 16, 26, 35], we assume the cloud computing platform has provided a secure channel for end users to offload their programs to the shared SSD. The related code-offloading techniques, such as secure RPC and libraries [22, 39, 72], have been deployed in cloud platforms [1, 48, 91]. However, an offloaded program can include (hidden) malicious code.

As for in-storage computing, we trust SSD vendors, who enable the execution of offloaded programs. We assume hardware vendors do not intentionally implant backdoor or malicious programs in their devices. However, as we deploy those computational SSDs in shared platforms (e.g., public cloud), we do not trust platform

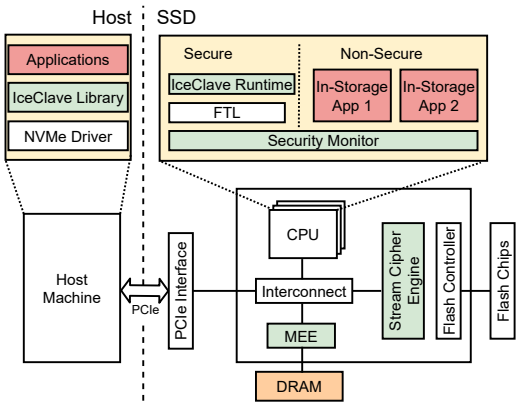


Figure 3: Overview of IceClave architecture.

operators who could initiate board-level physical attacks such as bus-snooping and man-in-the-middle attacks, or exploit the host machine to steal or destroy data stored in SSDs. Similar to the threat model for SGX, we exclude software side-channel attacks such as cache timing, page table side-channel attacks [62], and speculative attacks [51] from the threat model, since many of these attack approaches are cumbersome in reality [25].

We rely on the Error-Correction Code (ECC) available in flash controllers [40, 83] for ensuring the integrity of flash pages. To defend against attacks from cloud computing platforms or malicious host OS, users are usually encouraged to encrypt their data before storing them in the SSD. However, their data would still be leaked during the in-storage computing procedure. Thus, we have a more conservative design for achieving the security goals of IceClave.

We believe our threat model is realistic. First, as system-wide shared resource, SSDs have been widely used by multiple applications. Existing in-storage computing frameworks have enabled end users to offload their programs into the SSD. Second, once program is offloaded to the SSD, the in-storage program will escape the control of the host OS and initiate attacks in the new execution environment. Third, our threat model considers the potential physical attacks initiated by untrusted platform operators.

To the best of our knowledge, this is the first TEE framework for in-storage computing. It aims to defend against three attacks: (1) the attack against co-located in-storage programs; (2) the attack against the core FTL functions; (3) the potential physical attack against the data loaded from flash chips and generated by in-storage programs.

4 DESIGN AND IMPLEMENTATION

We present a TEE for in-storage computing with minimal performance and hardware cost. We show the overview of IceClave architecture in Figure 3. To achieve our goal, we propose to extend ARM TrustZone to create secure and normal world for security isolation and protection of different entities in FTL, while enabling memory encryption and verification with memory encryption engine (MEE).

4.1 Challenges of Building IceClave

To develop IceClave, we have to overcome three challenges.

- First, as SSD is shared by multiple applications, we need to ensure proper security isolation. Specifically, we need to not only enforce security isolation between in-storage applications and FTL functions, but also the isolation between applications and IceClave runtime (§4.2 and §4.3).
- Second, to protect data of in-storage programs at runtime, IceClave needs to ensure the data security whenever the user data leaves the flash chips (§4.4).
- Third, SSD controller has limited resource, such as DRAM capacity and processing capability; therefore, IceClave should be lightweight and not significantly affect the performance of in-storage applications (§4.5 and §4.6).

In the following sections, we will discuss how we address each of these challenges in details, respectively.

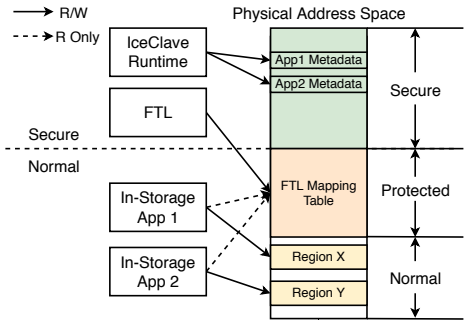


Figure 4: Memory protection regions in IceClave.

4.2 Protecting Flash Translation Layer

As FTL manages flash blocks and controls how user data is mapped to each flash page, its protection is crucial. If any malicious in-storage programs gain control over it, they can read, erase, or overwrite data from other users, which can cause severe consequences, such as data loss and leakage. And IceClave runtime manages how each in-storage application is initialized inside SSD, and maintains their metadata, such as in-storage program identity. If any in-storage program gains access to the metadata, the adversary can easily compromise the security of other in-storage programs.

To protect FTL and IceClave runtime from malicious in-storage programs, we need to ensure memory protections for different entities in the SSD. Specifically, we have to guarantee offloaded applications cannot access memory regions used by FTL and IceClave runtime. We also need to ensure offloaded applications cannot access each other's memory regions without proper permissions.

To achieve this, a straightforward way is to use TrustZone to create secure and normal worlds, and then place FTL functions and IceClave runtime in the secure world, and place all in-storage applications in the normal world. However, this will cause significant performance overhead for in-storage applications. This is because when an application accesses a flash page each time, it needs to context switch to the secure world which hosts the FTL and its address mapping table. Similarly, with the SGX-like approach, we can place FTL and in-storage programs in different enclaves, it will also generate significant performance overhead, as we have to switch frequently from one enclave to another.

To address this challenge, we partition the entire physical main memory space into three memory regions: normal, protected, and secure by extending TrustZone. We show these memory regions

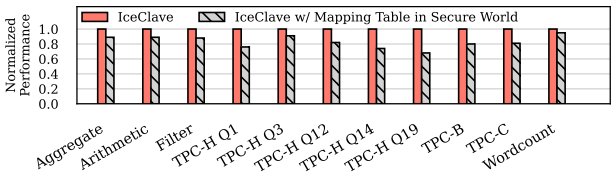
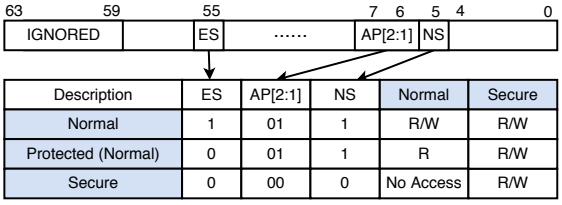


Figure 5: Performance comparison between IceClave and IceClave with FTL mapping table in secure world. Performance is normalized to IceClave.



Description	ES	AP[2:1]	NS	Normal	Secure
Normal	1	01	1	R/W	R/W
Protected (Normal)	0	01	1	R	R/W
Secure	0	00	0	No Access	R/W

Figure 6: In-storage memory protection in IceClave.

in Figure 4. Specifically, we allow FTL and IceClave runtime to execute in the secure world. And they have read/write permission to access the entire memory space. This is necessary, since the core functions of FTL need to manage the address mapping table, and IceClave runtime needs to manage each in-storage application, such as its TEE creation and deletion. We place in-storage applications in the normal world; therefore, they cannot access any code or data regions that belong to the FTL or IceClave runtime.

For the protected memory region in the normal world, we use it to host the shared address mapping table, such that in-storage applications can only read the mapping table entries for address translation, without paying the context-switch overhead. Figure 5 shows this optimization can improve the performance of in-storage applications by 21.6% on average, compared to the scheme with the FTL mapping table in the secure world. We will discuss the details of the flash access procedure and associated protection in § 4.6.

We demonstrate the details of the memory region attributes in Figure 6. Following the MMU specification of ARMv8 [9], we use the non-secure (NS) bit to indicate whether the memory access is performed with secure or normal right. We utilize the access control flags (AP[2:1]) and a reserved bit (ES bit in Figure 6) to create the protected region, in which IceClave gives read-only permission to the normal world and read/write permission to the secure world. It is worth noting that the in-storage memory protection can be easily implemented in an older version of ARM processors [7] by specifying the access control flags AP[2:0] as well as other processors such as RISC-V (see the discussion in §4.7).

4.3 Access Control for In-Storage Programs

Although each in-storage program only has the read access permission when accessing the mapping table of the FTL, a malicious in-storage program could probe the mapping table entries (e.g., by brute-force) that are managing the address translation for the data belonging to other in-storage programs. Henceforth, adversaries can easily access the data of other programs.

To address this challenge, we extend the address mapping table of FTL. We use the ID bits in each entry (8 bytes per entry) to track the identification of each in-storage TEE, and use them

Table 1: In-storage workload characterization.

Workload	Write Ratio	Workload	Write Ratio
Arithmetic	2.02×10^{-4}	TPC-H Query 1	6.40×10^{-6}
Aggregate	2.08×10^{-4}	TPC-H Query 3	3.96×10^{-3}
Filter	1.71×10^{-4}	TPC-H Query 12	2.99×10^{-5}
TPC-B	5.19×10^{-2}	TPC-H Query 14	3.94×10^{-6}
TPC-C	9.05×10^{-2}	TPC-H Query 19	9.92×10^{-7}
Wordcount	4.61×10^{-1}		

to verify whether an in-storage TEE has the permission to access the mapping table entry or not. The permission checking of flash accesses is performed with a dedicated process. It receives flash access requests from in-storage applications and performs permission checks before issuing the requests to the flash chips. This process has exclusive access to the flash chips in the normal world, which prevents unauthorized flash accesses from a malicious in-storage program. We use four bits for the ID by default, which introduces small (6.25%) storage cost to the mapping table. IceClave will reuse the ID for newly created TEEs within its runtime, and set the ID bits in the mapping table upon TEE creation (see the details in §4.6).

Each in-storage program only has accesses to the address mapping table of the FTL and allocated memory space. Accesses to other memory locations will result in a fault in the memory management unit. To further enhance the memory protection for in-storage programs, we also enable memory encryption and verification.

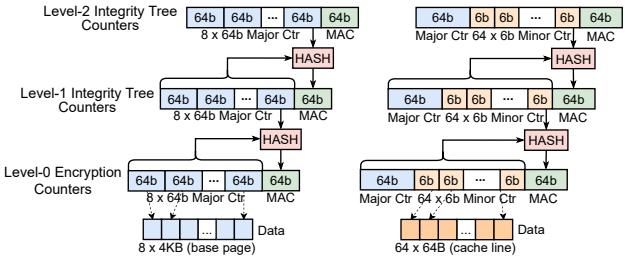
4.4 Securing In-Storage DRAM

In-storage programs load data from flash chips to the SSD DRAM for data processing. To conceal the data read from flash chips, IceClave secures the data transfer procedure by encrypting the accessed data before it is transmitted on the internal bus. Modern SSDs have employed dedicated encryption engine [31], however, it is a cryptography co-processor mainly used for full-disk encryption. In this work, we develop a lightweight stream cipher engine in the SSD controller for securing the data transfers from flash chips to the storage processor (see the implementation details in §5).

Although we enable the data encryption as we transfer data between SSD DRAM and flash controllers, the user data that includes raw data, intermediate data, and produced results could still be leaked at runtime. To address this challenge, IceClave enables both memory encryption and integrity verification.

Memory Encryption. The goal of memory encryption is to protect any data or code in a memory access from being leaked. To achieve this, a common approach is to encrypt the cache lines in the processor, when they are being written to memory. The state-of-the-art work usually uses split-counter encryption [12, 80, 90]. It works by encrypting a cache line through an XOR with a pseudo one time pad (OTP), and OTP is generated from encrypting a counter through a block cipher such as AES. The counter is incremented after each write back to guarantee temporal uniqueness. It is encoded as a concatenation of a major counter and a minor counter. When a minor counter overflows, the major counter is incremented, and all other minor counters are reset. The associated memory blocks also need to be re-encrypted. Therefore, such an encryption scheme has significant performance overhead.

This is less of a concern for in-storage computing because it is read intensive. We conduct a study of typical in-storage applications (see Table 4), and profile their number of memory accesses when



(a) Merkle tree for read-only pages (b) Merkle tree for writable pages

Figure 7: Memory encryption and integrity trees in IceClave.

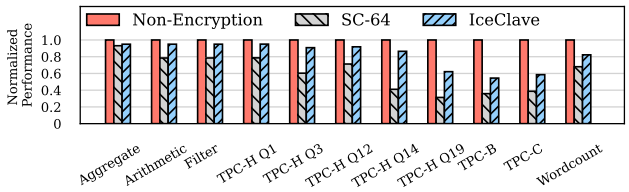


Figure 8: Performance comparison of Non-encryption, Split Counters (SC-64), and IceClave. It is normalized to the scheme without memory encryption.

running them (see the experimental setup in §6.1). Our observation is that most of these applications have a trivial portion of memory writes (see Table 1). These writes are usually caused by the produced intermediate data of in-storage programs at runtime. Based on this observation, we design a hybrid-counter scheme.

Hybrid-Counter Scheme. The key idea is that we only use major counters for read-only pages, and for writable pages, we apply the traditional split-counter scheme. As the minor counters will not change as long as the pages are read-only, we do not need minor counters for read-only pages. In this case, we can improve the caching performance for in-storage applications by packing more counters (eight for read-only pages) per cache line.

The hybrid-counter scheme maintains two types of counter blocks: split-counter blocks for writable pages, and major-counter blocks for read-only pages, as shown in Figure 7. We use two integrity trees to store the two types of counter blocks respectively. Although this requires slightly more memory space (0.01% of 4GB DRAM capacity)[1] to store the integrity trees, and needs two processor registers to store the two root message authentication codes (MACs) for integrity verification, the hybrid-counter scheme delivers improved performance by 43% on average (see Figure 8) for in-storage programs, compared to the current split-counter scheme.

As we use the hybrid-counter scheme in IceClave, we utilize the read/write permission bit in the page table entries to decide which counter blocks should be accessed. We also support dynamic permission changes of each memory page in IceClave. Specifically, for a read-only page, its corresponding counter is stored in the major-counter tree. When the page becomes writable and is updated, its corresponding major counter is incremented and copied to the corresponding entry in the split-counter tree, and also, the minor counters in that entry are initialized. At the same time, the page is re-encrypted using the new split-counter entry, which will be used for later accesses. When a writable page becomes read-only, its corresponding major counter is incremented and copied back to the major-counter tree. In-storage programs can use the memory protection mechanisms offered by ARM processors (see Figure 6) to update the permissions of memory pages. For example, for the memory region used to store the input for in-storage programs, its pages are set to be read only; for the memory region allocated for storing intermediate data, its pages are set to be writable.

Memory Integrity Verification. To ensure the processor receives exactly the same content as it wrote in the memory most recently, a MAC is generated for each memory block by hashing its data and encryption counter. On each memory access, the MAC is re-computed using the data and encryption counter, and compared against the stored MAC, such that any changes on the data or counter can be detected. The integrity tree also prevents replay attack which can roll back the data and MAC to their older versions. As shown in Figure 7, an integrity tree organizes MACs in a hierarchy, and the parent MAC ensures the integrity of its child MACs. The root of the tree is securely stored in the processor chip. When a cache line is written back to the memory, the merkle tree will update all the nodes on the path from the data block to the root.

In IceClave, we employ Bonsai Merkle Tree (BMT) [65]. It generates its first-level MAC by hashing counter blocks instead of data blocks. As discussed, IceClave maintains two Merkle trees, but the extra memory cost is negligible, compared to the traditional BMT.

4.5 IceClave Runtime

In this section, we discuss how IceClave runtime facilitates the execution of in-storage TEEs. It provides the essential functions for managing in-storage TEEs, such as TEE setup, TEE lifecycle and metadata management, and the interaction with the secure world. IceClave runtime also interacts with IceClave library deployed in the host machine. Note that IceClave library only exposes basic offloading interfaces (e.g., RPC) to end users. This not only reduces the trusted computing base but also simplifies the development of in-storage programs. We list the APIs of IceClave in Table 2.

IceClave allows a user to interact with the SSD using two APIs: `OffloadCode` and `GetResult`. Once the program is offloaded to the SSD, IceClave runtime will execute `CreateTEE()` to create a new TEE. At the same time, it will call `SetIDBits()` to set the ID bits (access permission, see §4.3) of the corresponding address mapping table entries in FTL with the list of logical page addresses specified by the in-storage program. According to our study on the popular in-storage programs, their code size is 28–528KB. However, for an offloaded program whose size is larger than the available space of SSD DRAM, the TEE creation will fail. During the execution of an in-storage program, `ThrowOutTEE()` will be called to handle program exceptions. IceClave runtime will abort the TEE for these cases that include (1) access control is violated, (2) TEE memory or metadata is corrupted, and (3) in-storage program throws an exception. Once the in-storage program is finished, IceClave runtime will call the `TerminateTEE()` to terminate the TEE.

With the assistance of TrustZone, IceClave supports dynamic memory allocation within each TEE. To avoid memory fragmentation, IceClave will preallocate a large contiguous memory region

[1]Given a 4GB DRAM, IceClave requires 0.5MB for the merkle tree used in Figure 7(a), and 4MB for the merkle tree used in Figure 7(b).

Table 2: IceClave API

API in IceClave Library	Description
OffloadCode (char* bin, uint* lpa, void* args, uint tid)	Invoke an offloading procedure specified by tid.
GetResult (uint tid, uint64_t* res)	Retrieve results from the offloaded program with tid.
API in IceClave Runtime	**Description**
CreateTEE (void* config, id_t &eid, tee_t* TEE)	Initiate a TEE and copies specified code into the TEE.
SetIDBits (const id_t &eid, uint64_t* lpns)	Set ID bits of the corresponding addr. mapping table entries.
TerminateTEE (tee_t* TEE)	Terminate the specified TEE, and reclaim resources.
ThrowOutTEE (tee_t* TEE, TEE_MSG* sm)	Abort the execution, and return an exception.
ReadMappingEntry (id_t &eid, uint64_t* lpa, uint64_t* ppa)	Request FTL to return the physical address.

(16MB by default). Upon TEE deletion, IceClave runtime will release the preallocated memory region.

4.6 Put It All Together

We illustrate the entire workflow of running an in-storage program with IceClave in Figure 9. Similar to existing in-storage computing frameworks [39, 72], IceClave library has a host-to-device communication layer based on PCIe, which allows users to transfer data between the host and SSD. As discussed in §3, we utilize the secure channel developed in modern cloud computing platforms for interactions between the host and shared SSD. The `OffloadCode` API (❶) described in §4.5 is called to offload programs. Its parameter `bin` represents the pre-compiled program in the form of machine code, and `lpa` is a list of Logical Page Addresses (LPAs) of data needed by the offloaded program. It uses task ID (`tid`) as an index for identifying the offloaded procedure.

IceClave runtime will create a new TEE for the offloaded program using `CreateTEE` (❷). At creation, IceClave runtime will allocate memory pages from the normal memory region to the TEE, while the TEE metadata will be initialized and maintained in the secure memory region. IceClave also executes `SetIDBits` to set access permissions in the address mapping table for LPAs. The TEE does not rely on FTL to get physical page addresses (see §4.2), as it can access the mapping table in the protected memory region (❸).

However, the in-storage program may occasionally encounter cache misses, when a mapping entry for the accessed LPA is not cached in the SSD DRAM. In this case, the TEE has to redirect the address translation request to the FTL via `ReadMappingEntry` (❹). This TEE will be paused and switched to the secure world, such that FTL will load the missing mapping table pages (❺), update the cached mapping table in the protected memory region, and return the PPA to the TEE. To avoid in-storage programs probing the entire physical space of the SSD, we enforce the access control (see §4.3). Any data on the data path to the TEE is encrypted with the stream cipher engine (❻). Note that users are encouraged to encrypt their data to defend against attacks from malicious host OS. They will send their decryption key to the TEE along with the offloaded program, and decrypt the data at runtime in the TEE. The TEE will be invoked by IceClave runtime once the in-storage program is readily prepared inside the TEE. IceClave runtime constantly monitors the status of initiated TEEs, secures memory regions, and ensures the mapping table permission guard. Exceptions will be thrown out if any aforementioned integrity is compromised.

During the entire lifecycle of a TEE, the hardware protection mechanisms described in §4.4 are enforced to prevent physical memory attacks. IceClave will also enforce strong isolation between TEEs and the FTL. The memory protection described in §4.2 will also

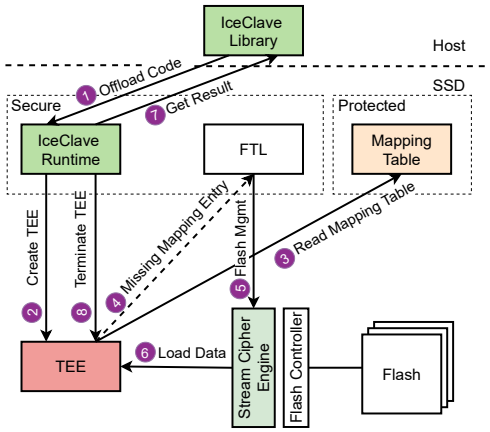


Figure 9: IceClave workflow of running in-storage programs. In this diagram, IceClave Runtime and Library, TEE, FTL, Stream Cipher Engine, and Flash Controller are compute regions. Mapping Table and Flash are memory regions.

be enforced in existing TrustZone memory controller (TZASC) [8]. Once reaching the end of the TEE program, the results are copied into the TEE's metadata region before terminating the TEE and reclaiming used resources (❽). IceClave will initiate a DMA transfer request to the host using NVMe interrupts, signaling the readiness of results. Results are returned to the host memory via `GetResult` (❼) provided in IceClave library.

In summary, IceClave can protect in-storage computing from both software and physical attacks with low overhead: (1) it enables memory encryption and verification for SSD DRAM with low overhead; (2) it protects shared FTL without frequent context switches between normal and secure worlds; (3) it protects transferred data from flash chips to SSD DRAM with an efficient stream cipher engine in the SSD controller.

4.7 Discussion and Future Work

In this paper, we exploit TrustZone technique in ARM processors to enable the memory protection between in-storage programs and FTL functions. This is driven by the fact that ARM processors are available in a majority of modern SSD controllers. As device vendors are also considering adopting the open-source RISC-V architecture in their controllers [32, 41, 73], the key idea of IceClave can also be implemented with new type of processors. To be specific, RISC-V defines three levels of privileges, including application level, supervisor level, and machine level [36]. We can map the normal, protected, and secure memory regions (see §4.2) to different memory regions in RISC-V respectively.

Table 3: Computational SSD simulator.

SSD Processor	ARM Cortex-A72 1.6GHz [10, 68]
Decoder Width	3 ops
Disp/Retire Width	5 ops
L1 I/D Cache	48KB/32KB
L2 Cache	1MB
SSD DRAM	DDR3 1600 MHz
Capacity	4GB
Organization	1 Channel, 2 Ranks/Channel, 8 Banks/Rank
Timing	t_{RCD}-t_{RAS}-t_{RP}-t_{CL}-t_{WR} = 11-28-11-11-12
Encryption Delay	AES-128: 60ns
SSD	1TB Flash-based SSD
Organization	8 channels, 4 chips/channel, 4 dies/chip 2 planes/die, 2048 blocks/plane 512 pages/block, 4KB page t_{RD}/t_{WR}=50/300μs
Bandwidth	600 MB/sec per channel

Beyond using the ARM and RISC-V processors to conduct in-storage computing, recent works also deploy hardware accelerators in SSD controllers [13, 24, 45, 46, 56, 57, 87]. They are also lacking the support of in-storage TEEs. We wish to extend IceClave to these in-storage hardware accelerators as future work.

5 IMPLEMENTATION DETAILS

Full System Simulator. We implement IceClave with a computational SSD simulator developed based on the SimpleSSD [38], Gem5 [37], and USIMM [21] simulator. This allows us to conduct the study with different SSD configurations, which cannot be easily conducted with a real SSD board. We use the SimpleSSD to simulate a modern SSD and its storage operations. We show the SSD configuration in Table 3. To enable in-storage computing in the simulator, we utilize Gem5 to model the out-of-order ARM processor in the SSD controller. We also implement the stream cipher in the integrated simulator to enable the data encryption/decryption as in-storage applications load data from flash chips. We use CACTI 6.5 [61] to estimate its hardware cost, and find that the cipher engine introduces only 1.6% area overhead to a modern SSD controller such as that of Intel DC P4500 SSD.

As IceClave will enable the memory verification in SSD DRAM, we leverage USIMM to simulate the DRAM in the SSD. We implement the Bonsai Merkle Tree (BMT) in the USIMM simulator, and use the hybrid-counter mode (see §4.4) as the memory encryption scheme. As discussed, the root of the integrity tree is stored in a secure on-chip register. The counter cache size is 128KB. To enable memory encryption and integrity verification, we enforce that each memory access will trigger the verification and update of the MAC and integrity tree. As we develop a full system simulator with Gem5, we run real data-intensive workloads, such as transaction database to evaluate the efficiency of IceClave in the following section.

Real System Prototype. To verify the core functions of IceClave, including TEE creation/deletion, FTL, and stream cipher engine, we also implement IceClave with an OpenSSD Cosmos+ FPGA board that has a Dual ARM Cortex-A9 processor [84]. We measure their overheads and show them in Table 5. We demonstrate the architecture of the stream cipher engine in Figure 10. Its key initialization block takes a symmetric key and an arbitrary initialization vector (IV) as the input to initialize the cipher. IceClave keeps the key in a secure register, while the IV can be public. Once initialized, the stream cipher generates 64 keystream bits per cycle. The generated

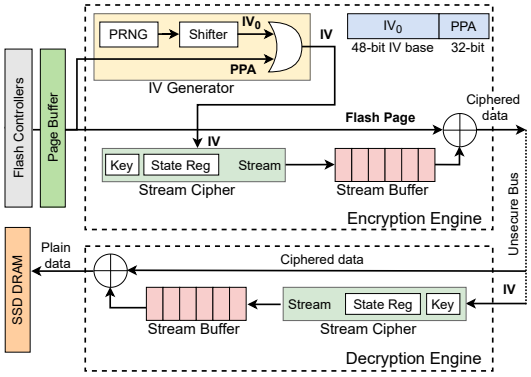


Figure 10: Stream cipher engine design in IceClave.

Table 4: In-storage workloads used in our evaluation.

Workload	Description
Arithmetic	Mathematical operations against data records
Aggregation	Aggregate a set of values with average operation
Filter	Filter a set of data that matches a certain feature
TPC-H Q1	Query pricing summary involving scan
TPC-H Q3	Query shipping priority involving join
TPC-H Q12	Query shipping modes and order priority with join
TPC-H Q14	Query market response to promotion with join
TPC-H Q19	Query discounted revenue with join and aggregate
TPC-B	Queries in a large bank with multiple branches
TPC-C	Online transaction queries in a warehouse center
Wordcount	Count the number of words in a long text [39]

keystream is XORed bitwise with the data read from flash chips to produce the ciphered data. The decipher uses the same key and IV to decode the ciphered data. The IV is constructed with temporally unique random numbers and spatially unique address bits. The orthogonal uniqueness enforces a strong guarantee that the same IV value will not be used twice during a certain period of time. The stream cipher algorithm we used refers to the Trivium [30]. To provide the uniqueness for different flash pages, we compose the IV by concatenating its physical page address (PPA) and the output of a pseudo-random number generator (PRNG).

6 EVALUATION

Our evaluation demonstrates that: (1) IceClave introduces minimal performance overhead to in-storage workloads while enforcing security isolation in SSD controllers (§6.2 and §6.3); (2) IceClave scales in-storage application performance as we increase the internal bandwidth of SSDs (§6.4); (3) It can benefit various SSD devices with different access latencies (§6.5); (4) IceClave still outperforms conventional host-based computing significantly while offering security isolation, as we vary the in-storage computing capability (§6.6 and §6.7); (5) IceClave enables concurrent execution of multiple in-storage programs with low performance overhead (§6.8).

6.1 Experimental Setup

We evaluate IceClave with a set of synthetic workloads and real-world applications as shown in Table 4. In the synthetic workloads, we use several essential operators in database system, including arithmetic, aggregation, and filter operations. As for the real-world applications, we run real queries from the TPC-H benchmark. Specifically, we use TPC-H Query 1, 3, 12, 14, and 19 that include the combination of multiple join and aggregate operations. In addition to these workloads, we also run write-intensive workloads TPC-B,

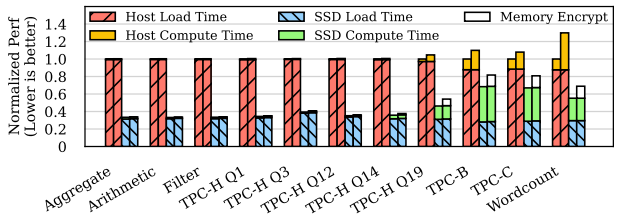


Figure 11: Performance comparison of Host, Host+SGX, ISC, and IceClave (from left to right). We show the performance breakdown of each scheme.

TPC-C, and Wordcount to further evaluate the encryption and integrity verification overhead of IceClave. In all these workloads, we populate their dataset (tables) to the size of 32GB, and place them across the channels in the SSD.

We compare IceClave with several state-of-the-art solutions. Particularly, we compare IceClave with the Intel SGX available in the host machine, in which we load the data from the SSD to the host memory and conduct the queries in the SGX. For this setting, we use a real server, which has an Intel i7-7700K processor running at 4.2GHz, 16GB DDR4-3600 DRAM, and 1TB Intel DC P4500 SSD. For fair comparison, we follow the specification of the Intel DC P4500 SSD to configure our SSD simulator. We also compare IceClave with current in-storage computing approaches that do not provide TEEs for offloaded programs. We list them as follows:

- **Host**: in which we load data from the SSD to the host memory, and execute the data queries using host processors. The host machine and SSD setups are described above.
- **Host+SGX**: in which we run data queries within the Intel SGX after loading data from the SSD. The version of the SGX SDK we use in our experiments is 2.5.101.
- **In-Storage Computing (ISC):** in which we run data queries with the ARM processors in the SSD controller, such that we can exploit the high internal bandwidth of the SSD.

6.2 Performance of IceClave

We show the normalized performance of running each query benchmark in Figure 11. We use the Host as the baseline, in which we run the query workload with the host machine while loading the dataset from the SSD. As shown in Figure 11, IceClave outperforms Host and Host+SGX by 2.31× and 2.38× on average, respectively. This shows that IceClave will not compromise the performance benefits of in-storage computing. As those data query workloads are bottlenecked by the storage I/O, the SGX on the host machine (Host+SGX) slightly decreases the workload performance. Compared to in-storage computing without security isolation enabled (ISC), IceClave introduces 7.6% performance overhead, due to the security techniques used in the in-storage TEE.

To further understand the performance behaviors of IceClave, we also demonstrate the performance breakdown in Figure 11. As for the Host and Host+SGX schemes, we partition their workflow into two major parts: data load and computing time. As we can see, Host+SGX incurs 103% extra computing time on average, caused by the SGX running in the host machine. As for the ISC and IceClave schemes, we profile the data load time from flash chips, the computing time with in-storage processors, and the overhead caused by the memory encryption and verification for IceClave. As shown in Figure 11, IceClave and ISC take much less time on loading time, as the internal bandwidth of the SSD is higher than its external bandwidth. And they require more time (2.47× on average) to execute the data queries. Compared to ISC, IceClave needs memory encryption and verification. For write-intensive workloads such as Wordcount, IceClave slightly increases memory encryption overhead. This is because Merkle tree intrinsically supports parallel updates, and our hybrid-counter design preserves this property by default. However, IceClave still outperforms host-based approaches significantly for a majority of in-storage workloads.

Table 5: Overhead source of IceClave.

Overhead Source	Average Time
TEE creation	95 μs
TEE deletion	58 μs
Context switch	3.8 μs
Memory encryption	102.6 ns
Memory verification	151.2 ns

Table 6: Extra memory traffic caused by memory encryption and verification when running in-storage workloads.

Workload	Encryption	Integrity Verification
Arithmetic	3.05%	2.27%
Aggregate	3.06%	2.26%
Filter	3.04%	2.26%
TPC-H Query 1	2.99%	2.22%
TPC-H Query 3	5.62%	4.5%
TPC-H Query 12	5.11%	3.78%
TPC-H Query 14	10.28%	5.39%
TPC-H Query 19	36.20%	24.75%
TPC-B	46.92%	36.68%
TPC-C	39.09%	31.72%
Wordcount	67.45%	43.81%

6.3 Overhead Source in IceClave

We also profile the entire workflow of running an in-storage program with IceClave. We show the critical components of IceClave and their overhead in Table 5. IceClave takes 95 μs and 58 μs (measured in real SSD FPGA board) to create and delete a TEE inside SSD, respectively. The overhead of context switch between secure world and normal world is 3.8 μs. As discussed in §4.2, IceClave has infrequent context switches at runtime, as it places the frequently accessed address mapping table in the protected memory region. The context switch happens mostly because the mapping table entries are missing in the protected memory region, and IceClave needs to switch to the secure world to fetch the mapping table from flash chips, and update them in the protected memory region.

Moreover, IceClave incurs much less memory encryption and verification operations, because most in-storage workloads are read intensive (see §4.4 and Table 1). The average execution times of each memory encryption and verification take 102.6 ns and 151.2 ns, respectively. We profile the additional memory accesses (see Table 6) incurred by fetching and overflowing counters in the memory encryption and integrity verification. We show the extra memory traffic in percentage in Table 6, when comparing to the regular memory traffic without enforcing memory security. Memory encryption and verification increase the memory traffic by 20.26% and 14.51% on average, respectively.

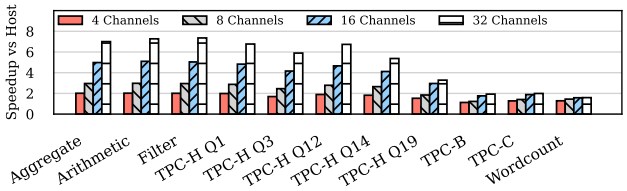


Figure 12: IceClave scales its performance, as we vary the internal SSD bandwidth by using different number of channels (normalized to Host).

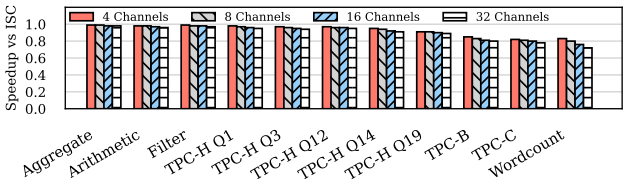


Figure 13: IceClave introduces minimal performance overhead (normalized to ISC), as we vary the internal SSD bandwidth by using different number of channels.

We also profile the number of flash address translations requested from a TEE, and find that only 0.17% of these address translations are missed in the cached mapping table in the protected memory region. This indicates that in-storage programs do not incur the context switch from the normal world to the secure world frequently, showing that IceClave is lightweight for in-storage workloads.

6.4 Impact of SSD Bandwidth

We now evaluate the performance sensitivity of IceClave, as we change different SSD parameters. We first vary the internal bandwidth of the SSD by changing the number of flash channels from 4 to 32. With this, the aggregated internal I/O bandwidth grows linearly, while the external bandwidth is capped by the PCIe bandwidth [18, 52]. We compare IceClave with the host-based approach (Host), and present the normalized speedup in Figure 12. For each data query workload, the performance benefit of IceClave scales significantly, as we increase the number of channels. To be specific, IceClave speeds up the performance by 1.7–5.0× over Host, showing that IceClave has negligible negative impact on the performance of in-storage computing. As for in-storage workloads that involve more complicated computations such as TPC-B, TPC-C, and Wordcount, increasing the internal bandwidth brings 1.2–1.8× performance speedup, and for others such as the synthetic workloads and TPC-H, IceClave obtain more performance benefits (1.9-6.2×). Note that IceClave achieves even more performance benefits than Host+SGX, because SGX introduces extra overhead (see Figure 11 and §6.2). And IceClave enables TEE for in-storage programs.

As we vary the internal SSD bandwidth, we also compare IceClave with ISC. As shown in Figure 13, IceClave decreases the application performance by up to 28% (8.6% on average), compared to ISC. Its additional overhead is slightly increased as we increase the number of channels for complicated data queries like TPC-C. This is mainly due to the increased overhead of memory encryption and integrity verification. However, IceClave offers a TEE for offloaded programs, making us believe it is worth the effort.

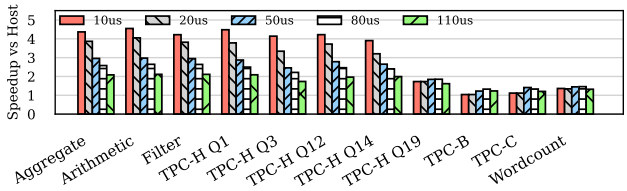


Figure 14: IceClave outperforms host-based computing for I/O-intensive workloads, as we vary the flash device latency.

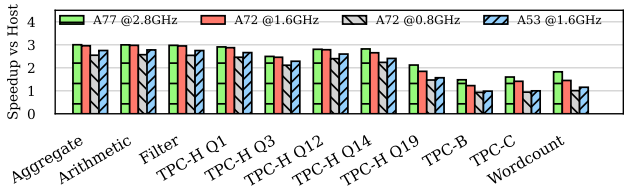


Figure 15: Sensitivity analysis of IceClave, as we vary the in-storage computing capability.

6.5 Impact of Data Access Latency

To understand how the data access latency affects the performance of IceClave, we vary the read latency of accessing a flash page from 10 μs, modeling an ultra-low latency NVMe SSD [2, 6], to 110 μs, modeling a commodity TLC-based SSD [60]. We keep the write latency as 300 μs, this is because most in-storage workloads are read-intensive, which involve few write operations to the dataset stored in the SSD. We use 8 channels in the SSD. We present the experimental results in Figure 14. As shown in Figure 14, compared to the host-based computing approach that is bottlenecked by the external PCIe bandwidth, IceClave delivers performance benefit (1.8–3.2×) for various SSD devices with different access latencies. For TPC-B, TPC-C, and TPC-H Q19 query workloads that require more computing resource for hash join operations, IceClave offers less performance benefit for the SSD with ultra-low latency, because the processors in the host machine provide more powerful computing resource.

6.6 Impact of Computing Capability

As we exploit embedded processors to run in-storage applications, it will be interesting to understand how the in-storage computing capability affects the efficiency of IceClave. We vary this parameter by using various models of embedded processor. We use our in-storage computing simulator to simulate the representative out-of-order (OoO) ARM processor A72, and the in-order processor A53 with different frequencies. We compare IceClave with the baseline Host that has an Intel i7-7700K processor running at 4.2GHz.

We show the normalized speedup in Figure 15. The performance of IceClave drops 13.7–33.4% as we decrease the CPU frequency of ARM processors. And an OoO processor A72 performs slightly better than the in-order processor A53 with the same CPU frequency. This demonstrates that IceClave can work with different type of ARM processors and deliver reasonable performance benefits.

6.7 Impact of DRAM Capacity in SSD

To evaluate the impact of the SSD DRAM capacity on the IceClave performance, we change the SSD DRAM size from 4GB to 2GB

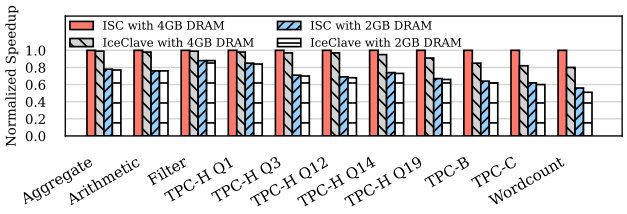


Figure 16: Sensitivity analysis of IceClave, as we vary the DRAM size in the SSD controller.

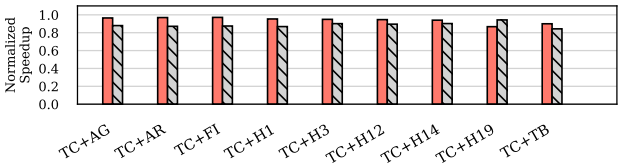


Figure 17: IceClave performance as we run two in-storage applications concurrently.

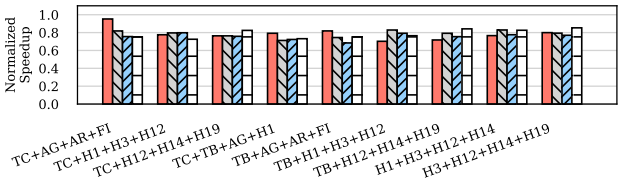


Figure 18: IceClave performance as we run four in-storage applications concurrently.

while using the same configurations as described in Table 3. We present the experimental results in Figure 16. As we decrease the SSD DRAM capacity, the performance of ISC drops by 12%–44%, as it has limited memory space to store its data set. The performance of IceClave follows the same trend. However, compared to ISC, IceClave still introduces minimal performance overhead.

6.8 Performance of Multi-tenant IceClave

To further evaluate the efficiency of IceClave, we run multiple IceClave instances concurrently, and each instance hosts one of the in-storage workloads as described in Table 4. We compare the application performance with the case of running each in-storage application independently without collocating with other instances. As shown in Figure 17, when we collocate the TPC-C instance with other workloads, the performance of in-storage applications is decreased by 6.1–15.7%. As we increase the number of collocated instances (see Figure 18), their performance drops by 21.4% on average. This is mainly caused by (1) the computational interference between the collocated IceClave instances, and (2) the increased cache misses (up to 8.7%) of the cached mapping table in the protected memory region. However, these in-storage programs still perform better than host-based approaches that are constrained by the external I/O bandwidth of SSDs.

7 RELATED WORK

In-storage Computing. In-storage computing has been intensively developed recently. Researchers have been exploring it for applications such as database query [33, 34, 47, 52], key-value store [72], map-reduce workloads [39, 47], signal processing [19], and scientific data analysis [81, 82]. To enable in-storage computing in modern SSDs, these prior works have developed various frameworks [39, 52, 66, 72]. However, most of them focus on the programmability and performance. Although there is still space for improvement in these aspects, such as having SSD array and filesystem support for in-storage computing [66], we have to overcome the security challenge of in-storage computing for its widespread deployment, since it poses threats to user data and flash devices. To the best of our knowledge, we are the first to propose building trusted execution environments for in-storage computing.

Trusted Execution Environment. To defend applications from malicious systems software, trusted hardware devices have been developed. A typical example is Intel SGX [17], which can create trusted execution environments for applications. Because of the enabled security isolation, SGX is extended or customized to support various computing platforms [5, 11, 27, 53, 78] and applications [14, 50, 63, 70]. The hardware devices with TPM [3] serve the similar purpose by utilizing the attestation available in commodity processors from AMD and Intel [58, 59, 77]. For ARM processors that are commonly used in mobile devices and storage controllers, they offer TrustZone that enables users to create secure world isolated from the OS [42, 69]. Unfortunately, none of these hardware devices can be directly applied to in-storage computing. The most recent work ShieldStore [50] and Speicher [14] applied SGX to key-value stores, however, none of them can protect the execution of in-storage programs. We develop specific trusted execution environments for in-storage applications.

Storage Encryption and Security. As we move computation closer to data in the storage devices, it would inevitably increase the trusted computing base, which poses security threats to user data. To protect sensitive user data while enabling near-data computing, a common approach is data encryption [64]. However, data leakage or loss would still happen at runtime, due to the lack of TEE support in modern SSD controllers. And adversaries can also initiate physical attacks to steal/destroy user data. An alternative approach is to enable computation on encrypted data [29, 93]. However, such an approach requires intensive computing resource, which cannot be satisfied by modern SSD controllers due to the limited resource budget [15, 39, 56, 85]. Our work IceClave presents a lightweight approach that can enforce security isolation for in-storage applications as well as defend against physical attacks.

8 CONCLUSION

Due to the lack of TEE support in SSD controllers, adversaries can intervene offloaded programs, mangle flash management, steal and destroy user data. To this end, we develop IceClave, a lightweight TEE which enables security isolation between in-storage programs and flash management. IceClave can also defend against physical attacks with minimal hardware cost.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and feedback. This work was partially supported by NSF grant CNS-1850317 and CCF-1919044, a gift fund from SK Hynix, and the Department of Defense under Contract FA8075-14-D-0002-0007, TAT 15-1158.

REFERENCES

[1] 2018. Amazon EC2 F1 Instances: Enable faster FPGA accelerator development and deployment in the cloud. https://aws.amazon.com/ec2/instance-types/f1/.

[2] 2018. Intel® Optane™ SSD DC P4801X Series. (2018).

[3] 2020. TPM 2.0 Library Specification. https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[4] Ahmed Abulila, Vikram S Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Providence, RI, USA.

[5] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *NDSS'18*.

[6] Anandtech. 2019. Memblaze's PBlaze5 X26: Toshiba's XL-Flash-Based Ultra-Low Latency SSD.

[7] ARM. 2007. ARM1156T2F-S Technical Reference Manual.

[8] Arm. 2013. Arm CoreLink TZC-400 TrustZone Address Space Controller. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0504c/DDI0504C_tzc400_r0p1_trm.pdf.

[9] ARM. 2020. ARM Architecture Reference Manual for ARMv8-A.

[10] ARM. 2020. ARM Storage. https://www.arm.com/solutions/storage.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA.

[12] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. 2019. Triad-nvm: Persistency for integrity-protected and encrypted non-volatile memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. 104–115.

[13] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. 2013. Intelligent SSD: A Turbo for Big Data Mining. In *Proceedings of the 22nd ACM International Conference of Information Knowledge Management (CIKM'13)*. San Francisco, CA.

[14] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *17th USENIX Conference on File and Storage Technologies (FAST'19)*. Boston, MA.

[15] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014).

[16] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[17] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[18] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Santa Clara, CA.

[19] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman. 2012. Active Flash: Out-of-core Data Analytics on Flash Storage. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST'12)*. Monterey, CA.

[20] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST'20)*. Santa Clara, CA.

[21] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. 2012. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep* (2012).

[22] Rakesh Cheerla. 2019. Computational SSDs. *Storage Networking Industry Association* (2019).

[23] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*.

[24] Benjamin Y. Cho, Won Seob Jeong, Doohwan Oh, and Won Woo Ro. 2013. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. In *Proceedings of the 1st Workshop on Near-Data Processing in Conjunction with the 46th IEEE/ACM International Symposium on Microarchitecture (WoNDP)*. Davis, CA.

[25] Victor Costan and Srinivas Devadas. [n. d.]. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf.

[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security'16)*. Austin, TX.

[27] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 857–874. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[28] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (USENIX Security'03)*.

[29] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. gonzalez, and Ion Stoica. 2020. Oblivious Coopetive Analytics Using Hardware Enclaves. In *Proceedings of European Conference on Computer Systems (EuroSys'20)*. Crete, Greece.

[30] Christophe De Canniere and Bart Preneel. 2005. Trivium specifications. In *eSTREAM, ECRYPT Stream Cipher Project*.

[31] Delkin Industrial. 2019. Encryption and Security Development in Solid State Storage Devices (SSD). https://www.delkin.com/blog/encryption-and-security-development-in-solid-state-storage-devices-ssd/.

[32] Western Digital. 2019. RISC-V: Accelerating Next-Generation Compute Requirements. https://www.westerndigital.com/company/innovations/risc-v.

[33] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. New York, NY.

[34] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. New York, NY, USA.

[35] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using Verification to Disentangle Secure-Enclave Hardware from Software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Shanghai, China.

[36] RISC-V Foundation. 2017. The RISC-V Instruction Set Manual. https://content.riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[37] gem5 development team. 2020. gem5 simulator.

[38] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. 2018. Amber*: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 469–481.

[39] B. Gu, A. S. Yoon, D. H. Bae, I. Jo, J. Lee, J. Yoon, J. U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. 2016. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. Seoul, Korea.

[40] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Washington, DC, USA.

[41] Gareth Halfacree. 2018. SiFive's RISC-V cores launch in two SSD families. https://www.bit-tech.net/news/tech/storage/sifives-risc-v-cores-launch-in-two-ssd-families/1/.

[42] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium (USENIX Security'17)*. Vancouver, BC.

[43] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*. Santa clara, CA.

[44] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Portland, OR.

[45] S. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*. Los Angeles, CA.

[46] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An Appliance for Big Data Analytics. *SIGARCH Comput. Archit. News* 43, 3 (June 2015).

[47] Y. Kang, Y. Kee, E. L. Miller, and C. Park. 2013. Enabling cost-effective data processing with smart SSD. In *Proceedings of the 28th IEEE Conference on Mass Storage Systems and Technologies (MSST'13)*. Lake Arrowhead, CA.

[48] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. 2018. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA.

[49] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA.

[50] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference (EuroSys'19)*.

[51] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland'19)*.

[52] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. Cambridge, Massachusetts.

[53] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys'18)*.

[54] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. Heraklion, Greece.

[55] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*.

[56] Vikram Sharma Mailthoday, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wenmei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. Columbus, OH.

[57] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA'19)*. Phoenix, AZ.

[58] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *2010 IEEE Symposium on Security and Privacy (Oakland'10)*.

[59] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An Execution Infrastructure for Tcb Minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'08)*.

[60] Micron. 2019. Micron 3D NAND Flash Memory.

[61] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. *HP laboratories* (2009).

[62] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. Boston, MA.

[63] C. Priebe, K. Vaswani, and M. Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (Oakland'18)*.

[64] Joel Reardon, Srdjan Capkun, and David Basin. 2012. Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security'12)*. Bellevue, WA.

[65] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 183–196.

[66] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 379–394. https://www.usenix.org/conference/atc19/presentation/ruan

[67] Gururaj Saileshwar, Prashant Nair, Prakash Ramrakhyani, Wendy Elsasser, Jose Joao, and Moinuddin Qureshi. 2018. Morphable counters: Enabling compact integrity trees for low-overhead secure memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 416–427.

[68] Samsung. 2020. SmartSSD Computational Storage Drive. https://samsungsemiconductor-us.com/smartssd/index.html.

[69] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. 2014. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.

[70] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy (Oakland'15)*.

[71] Security Flaws Found in Intel Software, Data Center SSDs. [n. d.]. https://www.tomshardware.com/news/intel-security-vulnerabilities-processor-diagnostic-tool-ssd,39845.html.

[72] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[73] Anton Shilov. 2019. Samsung to Use SiFive RISC-V Cores for SoCs, Automotive, 5G Applications. https://www.anandtech.com/show/15228/samsung-to-use-riscv-cores.

[74] David Shue and Michael J. Freedman. 2014. From Application Requests to Virtual IOPs: Provisioned Key-Value Storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. New York, NY, USA.

[75] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA.

[76] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin Levandoski, and David Lomet. 2015. Schema-Agnostic Indexing with Azure DocumentDB. *Proceeding of VLDB Endow.* 8, 12 (Aug. 2015).

[77] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. 2011. Logical Attestation: An Authorization Architecture for Trustworthy Computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP'11)*.

[78] Stavros Volos and Kapil Vaswani and Rodrigo Bruno. 2018. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. Carlsbad, CA.

[79] L. Szekeres, M. Payer, T. Wei, and D. Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (Oakland'13)*.

[80] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing paging overheads in SGX with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 665–678.

[81] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active Flash: Towards Energy-efficient, In-situ Data Analytics on Extreme-scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. San Jose, CA.

[82] Devesh Tiwari, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Simona Boboila, and Peter J. Desnoyers. 2012. Reducing Data Movement Costs Using Energy Efficient, Active Computation on SSD. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems (HotPower'12)*. Hollywood, CA.

[83] Hung-Wei Tseng, Laura M. Grupp, and Steven Swanson. 2013. Underpowering NAND Flash: Profits and Perils. In *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*.

[84] Hanyang University. 2020. The OpenSSD Project: Open-Source Solid-State Drive Project for Research and Education.

[85] Xiaohao Wang, You Zhou, Chance C. Coats, and Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Dresden, Germany.

[86] Steve Weis. 2014. Protecting Data In-Use from Firmware and Physical Attacks. *Proceedings of Black Hat* (2014).

[87] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA'20)*.

[88] What is a buffer overflow? And how hackers exploit these vulnerabilities. [n. d.]. https://www.csoonline.com/article/3513477/what-is-a-buffer-overflow-and-how-hackers-exploit-these-vulnerabilities.html.

[89] Bruce Wile. 2014. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. *White Paper* (Sep 2014).

[90] Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving cost, performance, and security of memory encryption and authentication. *ACM SIGARCH Computer Architecture News* 34, 2 (2006), 179–190.

[91] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. 2017. The Feniks FPGA Operating System for Cloud Computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*.

[92] Ning Zhang, Junichi Tatemura, Jignesh Patel, and Hakan Hacigumus. 2014. Re-Evaluating Designs for Multi-Tenant OLTP Workloads on SSD-BasedI/O Subsystems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*.

[93] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. Boston, MA.