Towards Isolated Execution at the Machine Level

Shu Anzai
The University of Tokyo
Tokyo, Japan

Masanori Misono
Technical University of Munich
Munich, Germany

Ryo Nakamura
The University of Tokyo
Tokyo, Japan

Yohei Kuga
The University of Tokyo
Tokyo, Japan

Takahiro Shinagawa
The University of Tokyo
Tokyo, Japan

ABSTRACT

Isolated execution with CPU-level protection, such as process sandboxes, virtual machines, and trusted execution environments, has long been studied to mitigate software vulnerabilities. However, the complexity of system software inevitably leads to vulnerabilities in isolated execution environments themselves, and the increase in hardware complexity makes it even more challenging to avoid hardware vulnerabilities. In this paper, we explore the possibility of isolated execution at the machine level using physically separated machines as an extreme case of isolation. We take advantage of recent hardware technologies to enable relatively low-latency communication between physical machines while dramatically reducing the attack surface and trusted computing base size compared to sharing computing resources on a single machine. As the first step in this direction, we discuss the security and performance of isolating processes to another machine with remote system calls and show its feasibility with preliminary experiments.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; • **Computing methodologies** → *Distributed computing methodologies*; • **Software and its engineering** → *Software architectures.*

KEYWORDS

Isolated Execution, RDMA, FPGA

ACM Reference Format:
Shu Anzai, Masanori Misono, Ryo Nakamura, Yohei Kuga, and Takahiro Shinagawa. 2022. Towards Isolated Execution at the Machine Level. In *13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22), August 23–24, 2022, Virtual Event, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3546591.3547530

1 INTRODUCTION

Security vulnerabilities have always been a serious problem in computer systems. Since vulnerabilities can cause critical damage to the systems, such as arbitrary code execution and information disclosure, it is desirable to eliminate as many vulnerabilities as possible. Unfortunately, despite significant efforts in vulnerability detection techniques [65, 94], modern computer systems are so large and complex that it is impractical to eliminate all vulnerabilities. In addition, attack methods are constantly evolving, ranging from classical memory corruption attacks [15, 18] to recent transient execution attacks [87], making the battle between attack and defense endless. Although fundamental resolutions such as the use of type-safe languages [46] and formal verification [37, 49] are becoming practical, there is still a considerable amount of codebase written in legacy programming languages, leading to the continuous discovery of security vulnerabilities [21]. Therefore, we need defense-in-depth mechanisms that assume the existence of vulnerabilities and can mitigate the damage from attacks.

Isolated execution is an effective technique to minimize the impact of security vulnerabilities. Following the principle of least privileges [67], separating the components of programs and confining them into protection domains will reduce the amount of information the attacker can access. Plenty of protection mechanisms for isolated execution have been studied even only recently, including intra-process isolation [68, 69, 81, 84], process-based sandboxes [16, 23, 28, 29], operating system (OS) containers [9, 51, 71, 78], intra-kernel protection [31, 32, 60, 61], virtual machines [47, 48, 58], and trusted execution environments (TEEs) [24, 42, 45, 93]. Unfortunately, these isolated execution environments are controlled by system software using primitive CPU functions, making it challenging to eradicate their own vulnerabilities [25, 64, 73, 74].

A generic and practical approach to addressing system software vulnerabilities is to reduce the attack surface [55] and keep the trusted computing base (TCB) small [67]. For example, various studies have attempted to limit the number of system calls and/or debloat (reduce) the amount of executable kernel code [6, 29, 33, 89, 91]. Similarly, hypervisors have also been studied in numerous attempts to reduce their attack surface and TCBs [8, 79], including those combined with library OSs [13, 40, 53]. However, these techniques still need privileged software to control the CPU-level protection mechanisms on the machine of isolated execution environments, leaving the attack surface and TCB on the same machine. In addition, recent increases in hardware complexity make it difficult to avoid attacks that exploit hardware vulnerabilities, such as transient execution [87] and RowHammer [59], making complete isolation on a single machine increasingly difficult.

This paper explores the possibility of isolated execution at the machine level as an extreme case of isolation. Taking the opposite approach to enhancing isolation on a single machine, we use physically isolated machines that do not share computational resources to achieve strong isolation. Unlike existing distributed computing, such as classical distributed operating systems [12, 80], current multi-tier cloud architectures [39], and recent serverless computing [50], we place most of the privileged software that controls isolated execution environments on a physical machine that is separated from the isolated execution environments themselves, with virtually no attack surface or TCB on the isolated execution machine. To enable secure and low-latency communication between physical machines, we exploit specialized FPGA-based hardware that allows unidirectional RDMA from a remote machine to any physical address on the target machine without software support.

We believe that this specialized RDMA hardware, coupled with the increasing difficulty of achieving vulnerability-free isolation at the CPU level, will bring machine-level isolation closer to a realistic option in balancing security and performance. However, there are still many unanswered questions regarding machine-level isolated execution, such as how to adapt existing software, what interface design is appropriate, and what the quantitative overhead is. As the first step toward answering these questions, we present a case study of machine-level isolation at a system call boundary, i.e., a sandbox system where untrusted applications run on a dedicated physical machine and system calls are handled by host processes and host OS on a physically separated machine. System call level isolation can leverage the existing isolation boundary of user and kernel space, facilitating the application of this approach to existing systems and reuse of cleanly designed system call interfaces.

To estimate the performance in this system call level approach, we conducted preliminary experiments using a micro-benchmark and two real-world applications, OpenSSL and SQLite. The experimental results show that this approach incurs a significant overhead on system calls themselves as expected, reaching 943 times, while OpenSSL has an overhead of at most 1.7%, and SQLite has an overhead of 2% on some workloads, although more than 100 times slower on some other workloads. These results indicate that this approach could be practical for some specific applications and workloads. We also conducted a security evaluation and clarified that this approach could achieve unprecedentedly strong isolation that can address various attacks, including hardware ones.

The remainder of this paper is organized as follows. Section 2 explains our threat model. Section 3 describes the design, and Section 4 describes the implementation. Section 5 presents the results of the performance evaluation, and Section 6 presents the security evaluation. Section 7 discusses related work, Section 8 discusses future work, and Section 9 summarizes this paper.

2 THREAT MODEL

We assume that an attacker can compromise a target application in the user-space process and completely take over its control. The attacker can issue arbitrary system calls and access arbitrary memory in the process's virtual address space, thereby attempting various attacks, including arbitrary code execution, memory corruption, and information leakage. For example, the attacker can attempt a return-to-user (ret2usr) attack, exploiting a vulnerability in the privileged code running on the same machine as the application to execute arbitrary code at the privileged level. We also assume that the application shares hardware resources such as CPU cache and physical memory with privileged code running on the same machine, so attackers can attempt attacks exploiting hardware vulnerabilities, such as transient execution or DRAM interference.

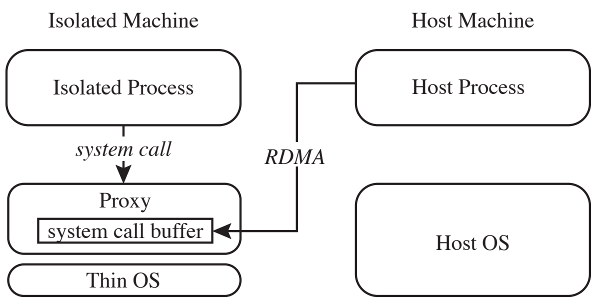

Figure 1: Overview of our architecture

We do not assume attacks that are complete within the OS kernel without executing arbitrary code placed in the application's address space. For example, time-of-check-to-time-of-use attacks to bypass permission checks by the OS kernel or denial-of-service attacks against the OS kernel through invalid arguments are outside the direct scope of this paper. We can mitigate such direct attacks on the OS kernel by combining existing techniques, which we will discuss in the next section. We also currently assume only one-way attacks from the compromised applications to the OS kernel; we assume that the OS kernel can be trusted and will not attack the applications. In future work, we plan to relax this assumption by running the applications in TEEs.

3 DESIGN

Our goal toward machine-level isolation in this paper is to create a robust sandbox system where untrusted applications can run without affecting other applications and OS kernels. Our sandbox shares many characteristics with previous process-based sandboxes using system call interposition [7, 11, 26, 27, 30, 36, 66]. However, separating the machine for untrusted applications from others will significantly improve the resistance to attacks that assume the CPU, memory, or physical devices are shared between untrusted applications and the target software.

Figure 1 shows an overview of our basic architecture. There are two physical machines: the *isolated machine* and *host machine*. The isolated machine runs *isolated processes*, that is, processes of untrusted applications. The host machine runs the *host process* that controls the isolated machine and works as a reference monitor. The two machines are connected through RDMA hardware.

When an isolated process issues a system call, a tiny piece of code, called the *proxy*, stores the number and arguments of the system call in the *system call buffer*. The proxy does not handle the system call but notifies the host process to handle the system call. The host process retrieves the contents of the system call buffer with RDMA, handles the system call using host OS functions as needed, and returns the result with RDMA. If necessary, the host process writes to the memory on the isolated machine with RDMA, e.g., to return I/O data or for memory management.

To reduce the attack surface and TCB of the isolated machine as much as possible, we run only a small privileged code, called the *thin OS*, on the isolated machine. Unlike a regular full-fledged OS, the thin OS provides only minimal functionality to set up execution environments for isolated processes. Specifically, the thin OS implements auxiliary functions that can only be performed on the CPU of the isolate machine, such as hardware initialization and configuration of the MMU and exception handlers. Also, to eliminate the need for drivers and agents for the thin OS, we use hardware that can perform RDMA without software support. In addition, we implement the thin OS using a type-safe language to eliminate vulnerabilities to memory corruption attacks. Since most of the system call logic is implemented within the host process, it is unlikely that the thin OS will be vulnerable to attacks or cause compatibility problems with applications.

To achieve complete mediation [67], we connect the isolated machine only to the host machine via RDMA so that system calls via RDMA become the only interface for isolated processes. In addition, we use *unidirectional* RDMA hardware that can initiate memory transfers only from the host machine, not from the isolated machine. This ensures that even if the isolated machine is compromised through the few interfaces present in the proxy or thin OS, all it can do is issue system calls to the host machine and cannot access host physical memory via RDMA.

In this architecture, isolated processes cannot launch attacks on the host machine that assume that hardware is shared, such as ret2usr, transition execution, and memory interference. Isolated processes may indirectly attack the host machine via system calls. However, the advantage of this architecture is that some of the complex system calls, such as process and memory management, can be implemented at the user level without host OS support, as they control hardware on the isolated machine rather than the host machine. Therefore, the host OS kernel is not compromised even if such system call implementation is compromised. System calls that would be transferred directly to the host OS, such as file and socket I/O, could still compromise the host OS kernel. However, we can mitigate this problem by combining existing techniques such as system call filtering [89], kernel debloating [6], and using unikernels [43]. Thus, this architecture is more secure than traditional sandboxes on a single machine.

We can configure the number of isolated processes, isolated machines, host processes, and host machines in several ways. Using one for each is the most isolated configuration but is less efficient in resource utilization. Placing multiple isolated processes belonging to the same security domain on a single isolated machine can balance security and resource utilization. In cloud environments, it would be reasonable to have multiple isolated machines with moderate performance for cases where high security is required, as in the case of bare-metal clouds. Multiple host processes could be placed on a single host machine to improve resource efficiency further. In this case, if the host process is compromised and the host OS kernel is further compromised, we cannot maintain isolation. Therefore, combining existing isolation mechanisms, such as more robust sandboxing or virtual machines, is desirable.

4 IMPLEMENTATION

In this section, we show our implementation for Linux x86-64 systems. We implemented the thin OS in Rust to eliminate memory-related vulnerabilities. The host process is currently implemented in C, but we will rewrite it in Rust in the future.

In the remainder, we first describe the RDMA hardware that we used, and then describe the implementation of the proxy, host process, and thin OS. Finally, we report the implementation status.

4.1 RDMA hardware

To summarize the description in Section 3, we need the following properties for the RDMA hardware.

Software independence: The host machine can perform RDMA to the isolated machine without driver software.

Full physical memory access: The host machine can read/write all the physical memory of the isolated machine.

Unidirectional access: Only the host machine can initiate RDMA to the isolated machine, not the other way around.

Notification capability: The isolated machine can notify the host machine of events.

Most of the current RDMA systems are designed for data communication in user space, and few meet all of the above requirements; they may require software for initialization and management on both machines, only allow access to pre-approved memory areas, or have difficulty limiting access to one direction. Therefore, we exploited FPGAs to create RDMA hardware that meets all the above requirements. Instead of implementing it from scratch, we take advantage of a part of the functionality already implemented in NetTLP [41]. NetTLP was originally developed for PCI Express (PCIe) device development to remotely handle transaction layer packets (TLP) of the PCIe bus in software. However, as part of its functionality, it has the RDMA capability that meets the above requirements, so we decided to reuse it.

The actual hardware of NetTLP is a PCIe device with FPGA and NIC. This PCIe device is connected to a remote machine via the NIC, and DMA read/write data can be transferred via the NIC. The DMA function of this device is remotely controlled by software on the remote machine, so the device can start DMA by itself without driver support and can function as unidirectional RDMA. The device itself appears to be a normal PCIe device, but software access to a part of the configuration space is sent to the remote machine via the NIC, so it can be used as a notification function.

4.2 Proxy

A proxy is a simple piece of code that mediates system calls from the isolated process and writes its information to the system call buffer in a predetermined physical address area. The information of a system call is the system call number and arguments, which are stored in CPU registers. For example, in Linux x86-64 ABI, the number of the `write(2)` system call is "2", so the number "2", the file descriptor, the buffer address, and its size are stored in the buffer.

After storing the system call information, the proxy notifies the host process of the issuance of the system call using the notification function of the RDMA hardware. Specifically, NetTLP hardware has the ability to notify a remote machine via NIC when the software accesses the physical memory region specified by the base address register (BAR) 4 in the configuration space. When the remote machine is notified, a callback function registered in advance to a library called libTLP is called. We use this callback function to allow the proxy to notify the host process. Note that we did not adopt the option to write the system call information directly to the BAR 4 area without using the in-memory system call buffer, because the one-word write to the BAR 4 area generates a callback each time, which incurs an extra overhead.

There are two possible places to implement proxies: kernel space and user space. In general, kernel-space implementations require a transition cost to kernel space while improving compatibility, and user-space implementations are the opposite. However, our current implementation requires access to the BAR 4 region of the NetTLP device. Therefore, this time we implemented the proxy in kernel space for compatibility, simplicity, and safety. However, it is possible to implement it in user space by using secure RDMA hardware with no extra features.

4.3 Host Process

The host process is an ordinary process that runs on the host OS. As described above, the host process registers a callback function for libTLP in advance so that it is called when the proxy notifies. In the callback function, the host process accesses the system call buffer in the isolated machine with read RDMA using the predefined physical address. Then, it inspects the system call number and determines the handling of the system call.

The subsequent behavior of the host process depends on the type of system call. For file and network access, the host process allocates a temporary buffer in the host process after performing a security check. In the case of a write or send system call, the host process first reads the data as a temporary buffer with RDMA from the buffer in the isolated process specified by the argument of the system call. At this time, since the argument of the system call is specified by the virtual address of the isolated process, the host process performs address translation from the virtual address to the physical address. The host process then issues the same system call to the host OS as specified and performs I/O. In the case of a read or receive system call, the host process writes data to the buffer of the isolated process with RDMA after the system call is returned. Finally, the host process writes the return value to the system call buffer with write RDMA.

For thread and memory management, such as `fork(2)`, `execve(2)`, and `mmap(2)`, the host process needs to manipulate the CPU contexts and page tables of the isolated process. Therefore, the host process cooperates with a small piece of proxy code to update the CPU registers and some data structures on the isolated machine. For simple administrative system calls, such as `getpid(2)`, the host process handles the system call directly without the help of the proxy or the host OS.

4.4 Implementation Status

Our implementation is still in an early stage; currently, we have implemented approximately 30 system calls, including `open(2)`, `read(2)`, and `mmap(2)`. However, these are enough to run simple applications, and we plan to implement more system calls to run more complicated applications.

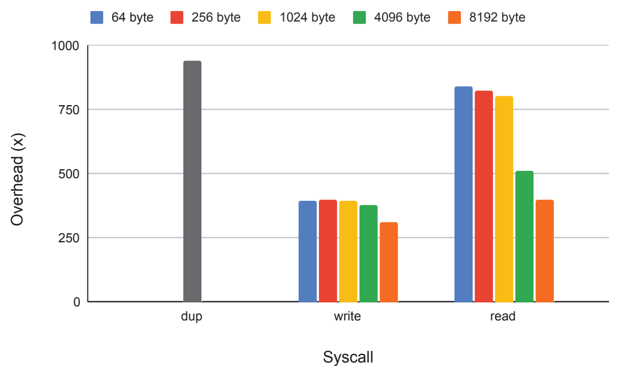

Figure 2: Overhead of syscalls

5 PERFORMANCE EVALUATION

This section presents experimental results of a preliminary performance evaluation of machine-level isolated execution of processes, using micro benchmarks and simple real-world applications. For comparison, we also measured the performance of processes in our environment and a typical single-machine environment.

We conducted our experiments using two machines with an Intel i7-9700 CPU and 32 GB of memory. We ran our thin OS on the isolated machine and Ubuntu 18.04 on the host machine. For the RDMA hardware, we used a Xilinx KC705 with a 16 Gbps PCIe Gen 2 4-lane link on the isolated machine and connected with the host machine via 10 Gbit Ethernet.

5.1 Micro benchmark

We implemented a micro-benchmark program that measures the number of system calls that can be executed in one second. We measured the performance of `dup(2)`, `read(2)` and `write(2)` system calls. Since the `dup(2)` system call could not be batched or cached [17], we can measure the pure overhead of executing the system call. We also measured the performance of I/O system calls using `read(2)` and `write(2)` system calls. In the `read(2)` and `write(2)` system call experiments, we measured 64, 256, 1024, 4096 and 8192 bytes of read and write.

Figure 2 shows the relative overhead of each system call in our environment compared to the single machine environment. The overhead in the `dup(2)` system call was 943.3x. It compares the cost of the CPU system call instruction versus the cost of the RDMA call, which is the worst-case overhead.

In both `write(2)` and `read(2)` cases, the overheads basically decreased as the byte size increased. In the single-machine environment, disk I/O latency increased, and the number of system calls per second decreased significantly as the byte size increased. It is also true in our environment, but since RDMA latency is much longer than disk I/O, system call performance does not decrease as much as in the single-machine environment, even with larger byte sizes. Thus, the relative overhead gradually decreases as byte size increases.

In our environment, the number of `read(2)` executed per second did not decrease as much as `write(2)` even when the

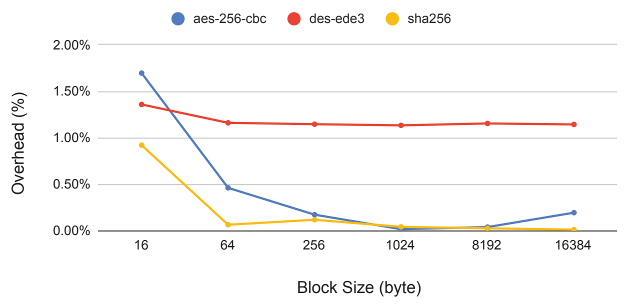

Figure 3: Overhead in OpenSSL

byte size increased. It is because `read(2)` performs write RDMA, which sends data without a completion message from the remote machine, while `write(2)` performs read RDMA, which requires a completion message from the remote machine after the data transfer. Therefore, as byte size increased, the overhead of `read(2)` decreased significantly compared to that of `write(2)`.

In the single-machine environment, the number of `write(2)` executed per second was lower than `read(2)` in general. It is due to our micro benchmark code. In this experiment, `write(2)` continues writing data to the same file, and `read(2)` continues reading data from the same file. Therefore, `write(2)` periodically needs to update metadata and flush data to disk in case of a crash, so `write(2)` was not executed as many as `read(2)`. On the other hand, the number of `write(2)` is not so different from the number of `read(2)` in our environment because of the high RDMA latency. As a result, the relative overhead was higher for `read(2)` than for `write(2)`.

5.2 Application benchmark

We evaluated the performance on two real-world applications, OpenSSL and SQLite.

5.2.1 OpenSSL. We evaluated the performance of OpenSSL in our environment using the `openssl speed` command. This command measures how much encryption can be done within a fixed time using various algorithms, and it is done on 16, 64, 256, 1024, and 8192 block sizes. We extracted the results of aes-256-cbc, des-ede3, and sha256. We linked the files statically since running dynamically linked files on our thin OS is unstable yet.

Figure 3 shows the result. As shown in the figure, little performance degradation occurs in all cases, and increasing block size did little to change the results. It is because few system calls are needed to execute OpenSSL.

This result indicates that when an application is CPU intensive and does not need to issue many system calls, our environment can achieve practical performance with higher security than a typical process environment on a single machine.

5.2.2 SQLite. We measured SQLite performance in our environment using the `speedtest1` [5] benchmark workload. This benchmark measures the time of a variety of operations in the database. Since our thin OS does not yet support multi threads, we compile SQLite and this benchmark using `-DSQLITE_THREADSAFE=0` and `-DSQLITE_OMIT_LOAD_EXTENSION` options and use the static link.

Figure 4 show the result. The overheads were quite different for different query identifiers. There are small overheads in 110, 120, 320, 400, 410, 500, 510, and 520, which are less than 10x. In particular, the lowest overhead is 1.02x (Query Identifier: 510). These queries are mainly REPLACE or SELECT. On the other hand, there are considerable overheads in 200, 240, 250, 270, 280, and 310, and they are more than 100x. Among them, the highest overhead is 263.15x (Query Identifier: 270). These queries are mainly DELETE or UPDATE.

6 SECURITY EVALUATION

The most important feature of our approach is the isolation of machines, which means the separation of CPU and memory between the user and kernel space. Therefore, our approach can prevent attacks assuming that the CPU or memory is shared between the user and kernel spaces. In this section, we discuss some case studies of attacks that our approach can prevent.

6.1 CPU attacks

Meltdown [52] or Spectre [38] are famous side-channel attacks that exploit speculative execution, and there are other similar attacks [2–4]. These attacks allow attackers to get sensitive data from the cache. Such attacks can be prevented by some mitigation techniques, such as using memory fence instructions appropriately. Still, it is difficult to address the root cause of the problem, and new side-channel attacks may be discovered in the future even if they are addressed.

Our approach can prevent such attacks. They are caused by sharing the CPU between user or kernel processes. In our approach, the isolated and host processes do not share the CPU. Thus, the cache on the CPU of the isolated machine contains only data from untrusted applications or that of the thin OS with little room to be attacked. Since the host machine's data do not reside in the CPU cache of the isolated machine, an attacker cannot take advantage of CPU characteristics to retrieve sensitive data.

6.2 Memory attacks

In our approach, the isolated process does not share physical memory with the host machine. Thus, we can prevent attacks that assume that the attacker's process and the target data reside in the same physical address space. An example of such an attack is arbitrary code execution through memory corruption attacks such as buffer overflows. For example, the ret2usr attack can exploit a kernel vulnerability to execute user code by hijacking the control flow [35]. CVE-2017-7308 [1] reports an example where the kernel does not check the data size, allowing a local user to gain kernel privileges or cause a DoS.

We can also prevent side-channel attacks on memory. For example, the RowHammer [59] attack exploits the electrical interaction between memory cells to destroy nearby data by successively accessing the same address in the DRAM. This attack can gain kernel privileges [70], but it is difficult to address it on a single machine fully. We can definitively prevent such an attack by separating the physical address space.

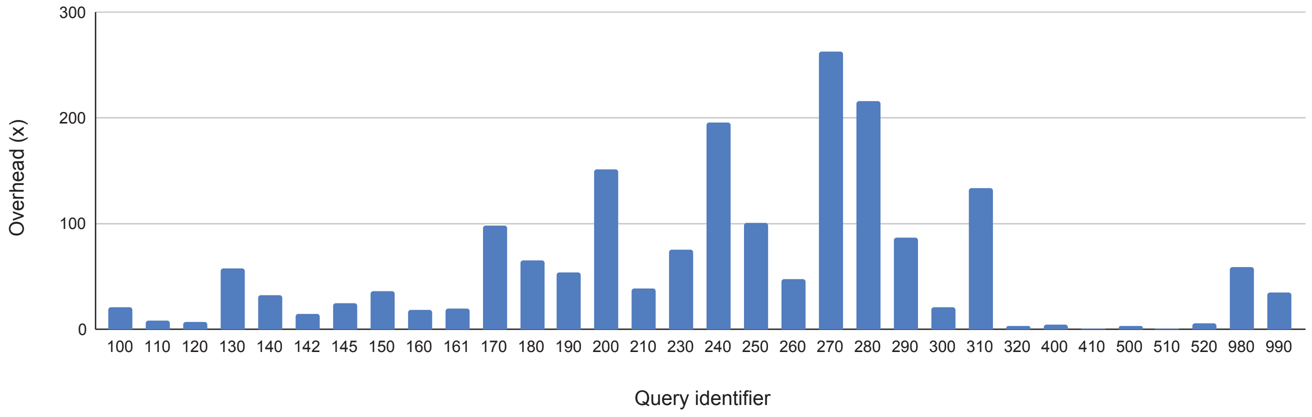

Figure 4: Overhead in SQLite

7 RELATED WORK

7.1 Isolated Execution

Numerous process-level sandboxes have been studied in the past [7, 11, 26, 27, 30, 36, 66]. These studies use OS-level mechanisms to intercept system calls and use OS-level protection, such as processes, to isolate and protect the reference monitor. However, these studies assume that we can trust OS-level isolation with CPU-level protection, and vulnerabilities in OS isolation mechanisms directly lead to the compromise of the reference monitor. In addition, process-level sandboxes are difficult to defend against side-channel attacks.

OS-level containers provide isolated execution environments for a set of processes. However, the isolation between containers relies on the host OS; the host OS is TCB, so if the host OS is compromised, the isolation is also compromised [28]. Although some studies improve container security by reducing the attack surface of the host OS [28, 83], it is still difficult to prevent attacks that exploit commonly used system calls.

VMs can achieve more robust isolation than containers [56]. However, as general-purpose hypervisors continue to grow in size, attack surfaces and TCBs are also increasing. Although there have been many studies to reduce the TCB size of the hypervisor [57, 77, 85], hypervisors of a certain size are still necessary to achieve isolation between VMs. In addition, countermeasures against side-channel attacks remain ongoing.

TEEs with CPU support, such as Intel SGX and Arm TrustZone, provide strong isolation [10, 63]. However, hardware-based TEEs also have various vulnerabilities due to their complexity. For example, Xu et al. [88] showed that a large amount of information could be extracted from an enclave using page faults. Bulck et al. [14] demonstrated that the untrusted OS could infer page access by an enclave without using page faults. Thus, achieving complete isolation at the CPU level has become difficult.

Co-processors are equipped in many practical systems and are also used in several studies to run additional security applications [76, 92]. Our approach is similar to these in that it offloads security-critical processing to a separate processor. However, our approach differs in using full-fledged CPUs connected via RDMA to run the main processing, such as applications and OS kernels, rather than additional processing.

Intel PKU helps realize in-process isolation [81, 82], which can provide isolated environments with a faster switch than inter-process isolation. However, Connor et al. [20] show vulnerabilities in existing PKU-based sandboxes. The study of Cerberus [82] shows that recent PKU-based schemes have security issues and address them in their new framework. Thus, establishing the security of PKU is still a work in progress.

DlibOS [54] exploits many-core processors with network-on-chip to provide strong isolation among applications and I/O stacks with low-latency communication. However, because DlibOS makes isolated programs coexist on processors that share a last-level cache and physical memory, protection against side-channel attacks such as transient execution and memory interference is not necessarily sufficient.

Distributed operating systems [12, 80] and serverless computing [50] allow for coordination between applications running on different machines. They can be considered to provide isolation between applications using physical machines but do not provide isolation between applications and system software.

7.2 TCB and Attack Surface Reduction

The OS and hypervisor fields have studied TCB and attack surface reduction.

One practical approach to TCB reduction is decomposing monolithic structures into microkernel-like structures. For example, in hypervisors, Xoar [19] breaks Dom0 of Xen functions into multiple components, each with a single purpose. Nexen [72] decomposes Xen and provides each VM with VM-slice, which has a part of the Xen functions and the corresponding private data. DeHype [86] decouples many of the KVM functions from the core part and runs them as libraries in each VM. HypSec [48] divides KVM into the small trusted corevisor and the other untrusted hostvisor. NOVA [77] decomposes the hypervisor and implements virtualization functions at the user level, like microkernels. The OKL4 microvisor [34] also implemented a microkernel-like hypervisor with para-virtualization. In the field of OS, PerspicuOS [22] takes the approach of nesting the kernel; the inner kernel is responsible for memory protection, and the outer kernel is treated as untrusted.

Another approach is to limit the purpose and reduce the functionality. CloudVisor [90] exploits nested virtualization to introduce a small security hypervisor under the existing hypervisor. Min-V [62] removes virtual devices that are not critical in cloud environments. Firecracker [8] targets container applications and removes functionalities such as legacy BIOS and device emulation, VM migration, and support for arbitrary kernels. LibraryOS and Unikernels [13, 40, 53] implement only the minimal functionalities necessary for specific applications. Kernel debloating [6, 33, 91] restricts access to the kernel for each application based on dynamic analysis.

Reducing sharing is also effective in reducing the attack surface. For example, Szefer et al. [79] have eliminated hypervisor attack surfaces by pre-allocating hardware resources. SPLIT-KERNEL [44] runs untrusted applications on a hardened kernel and trusted applications on a regular unmodified kernel.

Unfortunately, these approaches remain privileged software on the same machine as isolated execution environments.

8 FUTURE WORK

We need to implement more system calls and evaluate the performance in more applications. While this paper focused on the evaluation of file I/O system calls, we need to measure the performance of system calls related to memory management such as `mmap(2)`. Also, although the current proxy implements some memory management functions for the host process, it is possible to implement most of them with RDMA alone. We need to analyze the impact of such division of roles between the proxy and the host process on performance and security.

Although the performance of the system call level isolation approach is promising for some applications and workloads, as shown in Section 5, the system call overhead is still orders of magnitude higher than traditional systems. Therefore, we need performance improvement techniques to overcome this overhead. Software-based techniques include asynchronous and batched system calls. FlexSC [75] has already proposed and proven the effectiveness of asynchronous and bached system calls with exception-less system calls, and we can apply similar techniques to our approach. Hardware-based approaches include improving RDMA performance through better FPGA implementation and utilizing new hardware connected via lower latency networks. Moreover, since there is no noise due to hardware interrupts on the isolated machine, we can expect further performance gains for some CPU-intensive applications by exploiting this characteristic.

Interface design can also have a significant impact on overhead. In this paper, we have attempted an approach that uses system calls as the interface between the isolated and the host machines, which has the advantage of simplicity and ease of application to existing systems. However, RDMA communication is still much slower than context switches in the CPU, and the performance is significantly degraded when system calls are issued frequently. Therefore, it is essential to use or design an interface that can reduce the frequency of calls as much as possible. We are currently experimenting with an approach that uses virtio as the interface between the isolated and host machines. However, designing a more appropriate interface is also future work.

Protection of the host OS is also an issue, as discussed in Section 3. Files and network I/O from the isolated process can be implemented by forwarding them directly to the host OS, but in this case, there is a risk of compromise if a vulnerability in the host OS is exploited. In this regard, in addition to applying existing filtering and debloat techniques, another approach is to implement OS functions in user space using a library OS.

Finally, this time we assume an attack from the isolated process to the host machine, but not from the host process to the isolated process. However, if the host machine is compromised, the data on the isolated machine cannot be protected because the host machine can easily read and write data on the isolated machine using RDMA. Therefore, to prevent such attacks from the opposite direction, we need protection mechanisms that can ensure confidentiality and integrity, e.g., by taking advantage of TEE-like environments. We are considering using AMD SEV or Intel TDX to run the entire physical machine as a TEE, which would solve the confidentiality issue.

9 CONCLUSION

This paper describes the design and implementation of a machine-level isolated execution environment with system call boundaries. We achieved stronger isolation than CPU-level protection by running untrusted applications on isolated machines and transferring system calls onto physically isolated host machines. By utilizing RDMA hardware with capabilities such as software independence, full physical address access, unidirectional access, and notification, we were able to achieve secure yet low-latency physical machine-to-machine communication, bringing the machine-level isolation environment closer to practical performance. Experimental results show that while the overhead of a single system call is still very high, the overhead for some workloads in SQLite is practical, and there is almost no overhead in OpenSSL. Future work will include further performance evaluation, performance improvement, interface design studies, host OS protection, and isolated process protection.

REFERENCES

[1] 2017. CVE-2017-7308. https://nvd.nist.gov/vuln/detail/CVE-2017-7308.
[2] 2018. CVE-2018-3620. https://nvd.nist.gov/vuln/detail/CVE-2018-3620.
[3] 2018. CVE-2018-3639. https://nvd.nist.gov/vuln/detail/CVE-2018-3639.
[4] 2018. CVE-2018-3693. https://nvd.nist.gov/vuln/detail/CVE-2018-3693.
[5] 2022. speedtest1.c. https://sqlite.org/src/file/test/speedtest1.c.
[6] Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2435–2452. https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar
[7] Anurag Acharya and Mandar Raje. 2000. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *9th USENIX Security Symposium (USENIX Security 00)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/9th-usenix-security-symposium/mapbox-using-parameterized-behavior-classes-confine
[8] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa

Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 689–703.

[10] 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)*. USENIX Association, Broomfield, CO, 267–283.

[11] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. 2000. Operating System Enhancements to Prevent the Misuse of System Calls. In *Proceedings of the 7th ACM Conference on Computer and Communications Security* (Athens, Greece) *(CCS '00)*. Association for Computing Machinery, New York, NY, USA, 174–183. https://doi.org/10.1145/352600.352624

[12] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. 1989. Lightweight Remote Procedure Call. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, New York, NY, USA, 102–113. https://doi.org/10.1145/74850.74861

[13] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E. Engelstad, and Kyrre Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257. https://doi.org/10.1109/CloudCom.2015.89

[14] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1041–1056. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck

[15] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (apr 2017), 33 pages. https://doi.org/10.1145/3054924

[16] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop* (Virtual Event, Republic of Korea) *(CCSW '21)*. Association for Computing Machinery, New York, NY, USA, 139–151. https://doi.org/10.1145/3474123.3486762

[17] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith. 1995. The Measured Performance of Personal Computer Operating Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 299–313. https://doi.org/10.1145/224056.224079

[18] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. 2021. Exploitation Techniques for Data-Oriented Attacks with Existing and Potential Defense Approaches. *ACM Trans. Priv. Secur.* 24, 4, Article 26 (sep 2021), 36 pages. https://doi.org/10.1145/3462699

[19] Patrick Colp, Mihir Nanavati, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. 2011. Breaking up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 189–202. https://doi.org/10.1145/2043556.2043575

[20] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[21] Mitre Corporation. 2022. CVE – Common Vulnerabilities and Exposures.

[22] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 191–206. https://doi.org/10.1145/2694344.2694386

[23] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 459–474. https://www.usenix.org/conference/raid2020/presentation/demarinis

[24] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HYBCACHE: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, USA, Article 26, 18 pages.

[25] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv.* 54, 6, Article 126 (jul 2021), 36 pages. https://doi.org/10.1145/3456631

[26] T. Fraser, L. Badger, and M. Feldman. 1999. Hardening COTS software with generic software wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No.99CB36344)*. 2–16. https://doi.org/10.1109/SECPRI.1999.766713

[27] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition.. In *NDSS*.

[28] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[29] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, USA.

[30] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (San Jose, California) *(SSYM'96)*. USENIX Association, USA, 1.

[31] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2021. *Fast Intra-Kernel Isolation and Security with IskiOS.* Association for Computing Machinery, New York, NY, USA, 119–134. https://doi.org/10.1145/3471621.3471849

[32] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. 2020. Harmonizing Performance and Isolation in Microkernels with Efficient Intra-Kernel Isolation and Communication. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, USA, Article 27, 17 pages.

[33] Zhongshu Gu, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2014. FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 491–502. https://doi.org/10.1109/DSN.2014.52

[34] Gernot Heiser and Ben Leslie. 2010. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the first ACM asia-pacific workshop on Workshop on systems*. 19–24.

[35] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 957–972. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kemerlis

[36] Taesoo Kim and Nickolai Zeldovich. 2013. Practical and Effective Sandboxing for Non-root Users. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 139–144. https://www.usenix.org/conference/atc13/technical-sessions/presentation/kim

[37] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[38] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[39] Donald Kossmann, Tim Kraska, and Simon Loesing. 2010. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 579–590. https://doi.org/10.1145/1807167.1807231

[40] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: Fast, Specialized Unikernels the Easy Way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[41] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. 2020. NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 141–155. https://www.usenix.org/conference/nsdi20/presentation/kuga

[42] Sandeep Kumar and Smruti R. Sarangi. 2021. *SecureFS: A Secure File System for Intel SGX.* Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/3471621.3471840

[43] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in Unikernel Clothing. In *Proceedings of the Fifteenth European Conference on*

Computer Systems (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. https://doi.org/10.1145/3342195.3387526

[44] Anil Kurmus and Robby Zippel. 2014. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1366–1377. https://doi.org/10.1145/2660267.2660331

[45] 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* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 38, 16 pages. https://doi.org/10.1145/3342195.3387532

[46] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (Mumbai, India) *(APSys '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 7 pages. https://doi.org/10.1145/3124680.3124717

[47] Dingji Li, Zeyu Mi, Yubin Xia, Binyu Zang, Haibo Chen, and Haibing Guan. 2021. TwinVisor: Hardware-Isolated Confidential Virtual Machines for ARM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 638–654. https://doi.org/10.1145/3477132.3483554

[48] Shih-Wei Li, John S. Koh, and Jason Nieh. 2019. Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1357–1374. https://www.usenix.org/conference/usenixsecurity19/presentation/li-shih-wei

[49] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. 2021. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1782–1799. https://doi.org/10.1109/SP40001.2021.00049

[50] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, BingSheng He, and Minyi Guo. 2021. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.* (dec 2021). https://doi.org/10.1145/3508360

[51] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 418–429. https://doi.org/10.1145/3274694.3274720

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[53] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[54] Stephen Mallon, Vincent Gramoli, and Guillaume Jourjon. 2018. DLibOS: Performance and Protection with a Network-on-Chip. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 737–750. https://doi.org/10.1145/3173162.3173209

[55] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 3 (2011), 371–386. https://doi.org/10.1109/TSE.2010.60

[56] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[57] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *2010 IEEE Symposium on Security and Privacy*. 143–158. https://doi.org/10.1109/SP.2010.17

[58] Zeyu Mi, Dingji Li, Haibo Chen, Binyu Zang, and Haibing Guan. 2020. (Mostly) Exitless VM Protection from Untrusted Hypervisor through Disaggregated Nested Virtualization. In *Proceedings of the 29th USENIX Conference on Security Symposium*. USENIX Association, USA, Article 96, 18 pages.

[59] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318

[60] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 269–284.

[61] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight Kernel Isolation with Virtualization and VM Functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 157–171. https://doi.org/10.1145/3381052.3381328

[62] Anh Nguyen, Himanshu Raj, Shravan Rayanchu, Stefan Saroiu, and Alec Wolman. 2012. Delusional Boot: Securing Hypervisors without Massive Re-Engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/2168836.2168851

[63] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. 2020. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 776–789. https://doi.org/10.1109/ISCA45697.2020.00069

[64] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (jan 2019), 36 pages. https://doi.org/10.1145/3291047

[65] Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L. Agba. 2021. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. *ACM Comput. Surv.* 54, 2, Article 25 (mar 2021), 42 pages. https://doi.org/10.1145/3432893

[66] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. 2005. Authenticated system calls. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 358–367.

[67] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[68] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the 31st USENIX Conference on Security Symposium* (Santa Clara, CA, USA). USENIX Association, USA.

[69] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. *Donky: Domain Keys – Efficient in-Process Isolation for RISC-V and X86*. USENIX Association, USA.

[70] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. *Black Hat* 15 (2015), 71.

[71] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/3297858.3304016

[72] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. 2017. Deconstructing Xen.. In *NDSS*.

[73] Rui Shu, Peipei Wang, Sigmund A Gorski III, Benjamin Andow, Adwait Nadkarni, Luke Deshotels, Jason Gionta, William Enck, and Xiaohui Gu. 2016. A Study of Security Isolation Techniques. *ACM Comput. Surv.* 49, 3, Article 50 (oct 2016), 37 pages. https://doi.org/10.1145/2988545

[74] Federico Sierra-Arriaga, Rodrigo Branco, and Ben Lee. 2020. Security Issues and Challenges for Virtualization Technologies. *ACM Comput. Surv.* 53, 2, Article 45 (may 2020), 37 pages. https://doi.org/10.1145/3382190

[75] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/flexsc-flexible-system-call-scheduling-exception-less-system-calls

[76] Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. In *Proceedings of the 2016 Network and Distributed System Security Symposium*. Internet Society.

[77] Udo Steinberg and Bernhard Kauer. 2010. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 209–222. https://doi.org/10.1145/1755913.1755935

[78] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security Namespace: Making Linux Security Frameworks Available to Containers. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1423–1439.

[79] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. 2011. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 401–412. https://doi.org/10.1145/2046707.2046754

[80] Chandramohan A. Thekkath, Henry M. Levy, and Edward D. Lazowska. 1994. Separating Data and Control Transfer in Distributed Operating Systems. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VI)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/195473.195481

[81] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[82] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[83] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. 2017. Mining Sandboxes for Linux Containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 92–102. https://doi.org/10.1109/ICST.2017.16

[84] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*. 592–607. https://doi.org/10.1109/SP40000.2020.00087

[85] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating Functions at the Hardware Limit with Virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 644–662. https://doi.org/10.1145/3492321.3519553

[86] Chiachih Wu, Zhi Wang, and Xuxian Jiang. 2013. Taming Hosted Hypervisors with (Mostly) Deprivileged Execution. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society. https://www.ndss-symposium.org/ndss2013/taming-hosted-hypervisors-mostly-deprivileged-execution

[87] Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *ACM Comput. Surv.* 54, 3, Article 54 (may 2021), 36 pages. https://doi.org/10.1145/3442479

[88] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *2015 IEEE Symposium on Security and Privacy*. 640–656. https://doi.org/10.1109/SP.2015.45

[89] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A GVisor Case Study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (Renton, WA, USA) *(HotCloud'19)*. USENIX Association, USA, 16.

[90] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 2011. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/2043556.2043576

[91] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. KASR: a reliable and practical approach to attack surface reduction of commodity OS kernels. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 691–710.

[92] Lei Zhou, Fengwei Zhang, Jidong Xiao, Kevin Leach, Westley Weimer, Xuhua Ding, and Guojun Wang. 2021. A Coprocessor-Based Introspection Framework Via Intel Management Engine. *IEEE Transactions on Dependable and Secure Computing* 18, 4 (2021), 1920–1932. https://doi.org/10.1109/TDSC.2021.3071092

[93] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-Based Verifiable Database. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 2182–2194. https://doi.org/10.1145/3448016.3457308

[94] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* (jan 2022). https://doi.org/10.1145/3512345