Flexible Hardware-Managed Isolated Execution: Architecture, Software Support and Applications

Dmitry Evtyushkin*, Jesse Elwell*, Meltem Ozsoy†, Dmitry Ponomarev*,
Nael Abu Ghazaleh‡, Ryan Riley§

*State University of New York at Binghamton,
{devtyushkin,jelwell,dima}@cs.binghamton.edu
†Intel Corporation, ‡University of California at Riverside, §Qatar University
meltem.ozsoy@intel.com, naelag@ucr.edu, ryan.riley@qu.edu.qa

Abstract—We consider the problem of how to provide an execution environment where the application's secrets are safe even in the presence of malicious system software layers. We propose Iso-X — a flexible, fine-grained hardware-supported framework that provides isolation for security-critical pieces of an application such that they can execute securely even in the presence of untrusted system software. Isolation in Iso-X is achieved by creating and dynamically managing compartments (isolated software modules) to host critical fragments of code and associated data. Iso-X provides fine-grained isolation at the memory-page level, flexible allocation of memory, and a low-complexity, hardware-only trusted computing base. Iso-X requires minimal additional hardware, a small number of new ISA instructions to manage compartments, and minimal changes to the operating system which need not be in the trusted computing base. The run-time performance overhead of Iso-X is negligible and even the overhead of creating and destroying compartments is modest. An FPGA implementation of Iso-X runtime mechanisms shows a negligible impact on the processor cycle time.

Index Terms—Security, Hardware security, Isolated execution.

1 Introduction

One of the challenges in securing today's computing systems is how to efficiently protect the critical parts of security-sensitive applications from attacks that are launched using untrusted or compromised system software layers. Modern operating systems (OS) and virtualization layers are growing into large and very complex pieces of code. Today's OS kernels and hypervisors are large pieces of code and it is virtually impossible to design them without exploitable vulnerabilities [54], [61]. Many recent attacks, exploiting these vulnerabilities, have been successfully demonstrated [10], [23], [24], [25], [41], [56], [75], [82].

One approach to providing a secure execution environment in the presence of malicious software layers uses the concept of isolated execution, where the security-critical pieces of application code execute in *isolated software modules* or *compartments* [9], [15], [17], [19], [20], [37], [38], [43], [45], [47], [67]. These modules are inaccessible to the system software layers and are managed either entirely by the hardware [15], [45], [47], [53] or by a special layer of secure software that is sometimes assisted by hardware [17], [19], [20], [38], [43], [67]. The idea of supporting secure isolated environments has also received considerable attention from industry, exemplified by ARM's Trustzone [6], IBM's SecureBlue++ [15], Intel's SGX [9], [21], and Amazon's CloudHSM service [1].

Isolation schemes that limit their trusted computing base (TCB) only to hardware are arguably more secure compared to the schemes that include software into the TCB. This is because hardware schemes are not susceptible to software attacks (such as a buffer overflows and subsequent code reuse attack) on the trusted software layers. Unless security-critical software layers are formally verified, it is dangerous to assume that such attacks are impossible. While the hardware itself can also be buggy, it is much more difficult to exploit bugs in the HDL code than in C/C++ code that is typically used to implement system software. In fact, there are no examples of such exploitation to the best of our knowledge, aside from some denial of service attacks [50]. In addition, depending on the frequency of using software-based checks, software solutions can have noticeable performance degradation. On the other hand, hardware solutions often lack flexibility, have high granularity of protection, or do not support full-fledged secure execution environment. This can lead to a number of limitations, such as imposing constraints on the number and size of protected software compartments, or restricting their functionality. We provide a detailed comparison of previous software and hardware security schemes, including their limitations, in Section 5.

In this paper, we propose Iso-X (Isolated eXecution) — a hardware-managed framework for supporting a fine-grained, flexible and full-fledged isolated execution environment. Iso-X relies on simple OS functionality only to support flexible allocation of memory, eliminating the restrictions inherent in prior hardware-only isolation schemes. As a result, Iso-X combines the benefits of hardware and software-managed designs in terms of security and flexibility.

Iso-X achieves the execution isolation through a series of techniques that center around the use of *secure compartment page tables* to dynamically map and maintain memory pages for the compartments. The isolation of compartments is accomplished with only six required additional ISA instructions for compartment management, as well as two optional instructions to support page swapping,

resulting in a simple hardware implementation. While the Iso-X design requires that the application code be written in a way that explicitly marks the security-sensitive code to be isolated, the remaining software layers incur minimal changes. The key advantages of Iso-X compared to Intel's SGX (the most closely related design to Iso-X) is that *Iso-X does not limit where in memory the compartment memory pages can reside*. Not only does this provide better flexibility in memory usage and performance advantages in memory-constrained situations, but also can protects against recently introduced page-level side-channel attacks [78] on compartments, as we detail in Section 5. This submission is an extension of the conference paper [32] that appeared in the International Symposium on Microarchitecture (MICRO), 2014. The contributions of this submission are the following:

- We describe the Iso-X security architecture — a hardware/software co-design that supports the execution of security-critical pieces of application code inside isolated compartments. The Iso-X system is built around the concept of per-compartment page tables, allowing flexible memory usage and protection against page-level side-channel attacks on compartments driven by the page faults.
- We present evaluation of an integrated HDL implementation of the runtime mechanisms Iso-X with an OpenRISC processor core [42]. The resulting design was synthesized onto an FPGA Altera DE0-Nano board. The results show only a 2% increase in the cycle time due to the Iso-X logic. We also demonstrate that the performance impact of Iso-X is negligible both in secure and non-secure execution mode.
- We present detailed security analysis of Iso-X. This material is contained in Section 4 and is a new contribution compared to the MICRO 2014 version of the paper.
- We overview software support that is needed to fully implement the working Iso-X system. This includes support from the system software, as well as modifications to the application code itself. This is a new contribution compared to MICRO 2014 paper and this material is described in Section 7.
- We present several example scenarios of using the Iso-X system. These examples include the cloud scenario, secure machine attestation mechanism, and also the application of compartments to reducing the costs of remote procedure calls. This Section (Section 8) has been significantly extended compared to MICRO 2014 paper.
- We provide a comparison of Iso-X with Intel's SGX architecture, including recently published SGX2 extensions [4]. This comparison considers the relationship of both architectures to recently introduced side-channel attack on isolated systems [78]. Discussion of the vulnerability to this side channel attack is the new contribution in this submission.
- We provide a more complete comparison with related work, including several recent studies that appeared after our MICRO 2014 paper was published.

2 Threat Model and Assumptions

We assume that any portion of the system software stack, including the OS and the hypervisor, can be potentially compromised. The software trusted computing base (TCB) in the Iso-X system is thus limited to the developer-defined security-critical code. Iso-X guarantees strong protection of the compartment's internal memory from any malicious OS or hypervisor activity. Our isolation mechanism does not prevent compartments from sending sensitive data to untrusted domains, for example by writing it into unprotected memory. The compartment code is considered to be responsible for protecting its sensitive data, while the hardware isolation mechanism guarantees sturdiness of the compartment's perimeter.

While Iso-X relies on some basic functionality of the OS, such a reliance does not compromise security. Even if an attacker tampers with the OS services that offer this functionality, it can only lead to denial of service, but never to the leakage of the compartment state.

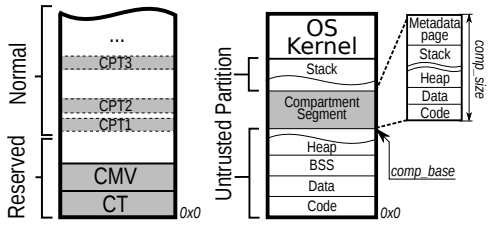
We assume that the hardware TCB of Iso-X is limited only to the microprocessor, memory controller, physical memory (DRAM), and system buses. We assume that all hardware in the TCB is implemented correctly and does not contain backdoors or bugs. In addition, we assume that hardware attacks (such as snooping on the memory bus or probing the physical memory) are not part of the threat model. We make this assumption for two reasons. First, hardware attacks are more difficult to perform than software attacks. Second, if the proposed architecture is deployed in a cloud environment, then it is reasonable to assume that a cloud operator will offer physical security of the system to protect its reputation. This is consistent with the assumptions made by recent works [31], [64], [70]. Recently some concerns have been expressed regarding the use of isolated execution environments with physical memory protection for Digital Right Management (DRM) and description copy-protected impossible to reverse engineer software (including malware) [57]. Therefore, in some cases the absence of physical memory protection can be viewed as an advantage.

We note that it is possible to amend Iso-X to consider physical memory to be untrusted, by incorporating well-known techniques for memory integrity verification and encryption [17], [68]. However, just with those previous designs, memory integrity checking logic will introduce significant performance and hardware overhead. In general, these techniques are orthogonal to the core isolated execution support, but they have to be used if the physical memory can not be trusted.

The current Iso-X design assumes the absence of vulnerabilities in compartment code. However, if such vulnerabilities are considered and can be used to launch, for example, code reuse attacks [62] additional techniques for protecting against such attacks will have to be implemented on top of Iso-X. Many such schemes have been proposed in recent literature [12], [26], [39], [51]. This is an orthogonal attack vector and will require separate solutions.

We do not explicitly consider architectural side and covert channel attacks [33], [34], [79] in this work. Processor caches can leak some sensitive information, such as memory access patterns, from compartments. Several techniques exist to protect against these attacks [28], [73] but further research is required to apply them directly to isolated execution environments, where the attacker can utilize the capabilities of a compromised operating system. However, Iso-X protects against recently proposed page-granularity side-channel attack on compartments [63], as we discuss in Section 5.

We do not consider denial-of-service (DoS) in our



(a) Physical memory layout (b) Virtual memory layout

Fig. 1: Iso-X Memory Layout

threat model, because it is already trivial for a malicious OS to deny service to an application or compartment.

3 Iso-X Design and Implementation

This section describes the Iso-X architecture and implementation.

3.1 Iso-X Design Overview

In Iso-X, application developers partition their programs into one Untrusted Partition (UP) and one or more Trusted Partitions (TP). The UP contains non-critical program code and data as well as all system software and libraries that are also assumed to be untrusted. The TPs contain security-critical code fragments and associated data, along with stack and heap memory regions to provide a fully-functional execution environment. The compartments also maintain a library for performing secure interaction with the rest of the system.

3.2 Protected Structures for Supporting Iso-X Compartments

Memory protection is at the core of the Iso-X design. Iso-X protects physical memory using two mechanisms. First, the basic data structures used for managing Iso-X are themselves stored in reserved memory that is only accessible by the Iso-X hardware. This restricted memory region is defined at boot time and it does not change during execution. Second, unreserved physical memory can also be dynamically protected at runtime for use by the individual compartments and for storing compartment-related metadata. The statically reserved memory holds two Iso-X structures:

Physical Page Compartment Membership Vector (CMV): This data structure is used to facilitate dynamic protection of memory pages. The CMV is a bit vector with one bit for each physical memory page in the system, specifying whether this page currently belongs to any compartment. The CMV is used to ensure that compartment pages and their metadata are never accessed by non-compartment code. It is also used to protect against double-mapping of compartment pages to other compartments, as described later. The CMV bits are also cached as part of the regular TLB entries, which are extended by a single bit that we call the Compartment bit, or the *C* bit.

Compartment Table (CT): The CT maintains the metadata that describes all compartments that have been created in the Iso-X system. It is indexed by the compartment ID, and the format of each CT entry is shown in Table 1. CPT refers to compartment page tables — another Iso-X data structure that is explained later in this section.

In addition to the static data structures described above, Iso-X also maintains dynamic structures that are established on-demand as compartments are created, used and destroyed. These structures reside in regular memory and get initialized as needed. They include:

Compartment Page Tables (CPT): The CPT (one for each compartment) maintains page address translations for compartments, similar to regular page tables. Although it duplicates information stored in regular page tables, the CPT is needed to protect the page mappings of the compartments from malicious modifications by the system software. Since the management of CPTs is a security-critical operation, it can only be manipulated by the hardware. Because implementing multi-level page tables completely in hardware is complicated, we chose to implement CPTs as single-level page tables. In general, a compartment will only use a portion of the virtual address space of the process to which it belongs. Therefore, a modest number of page table entries are often sufficient to manage the compartment memory. Please note that Iso-X does not impose restrictions on the size of the compartment's virtual address space. This size is determined by the size of the CPT structure, the storage for which is provided by the OS during the compartment initialization process. Compartment users in Iso-X should avoid requesting very large virtual address spaces, as this will result in excessive memory usage for the CPT pages. For instance, a single 4KB CPT page can service 2MB of compartment memory, assuming that each CPT entry is 64-bits wide. To address such limitation, Iso-X can use 2MB pages. In this case, a single CPT page will service about half a terabyte of compartment memory.

In addition, each CPT entry also maintains storage for hash values to support optional swapping, this is described in more detail in Section 3.10.

Compartment Metadata Page: Each compartment also maintains a special page called the Compartment Metadata Page. This page is used for storing Iso-X specific data structures inside of the compartment memory space at a predefined location, such as the first compartment page. The information that needs to be stored there includes: the context data of the compartment, the certificate of the compartment, and the compartment public key. The metadata page is protected from the OS just as any other compartment page. Section 3.5 presents more details on how this data is used.

In addition to the memory structures described above, Iso-X also requires minor modifications to the on-chip hardware. First, we add a small hardware structure (called **CCR** — Current Compartment Register) that is composed of two parts: the **CCR.CT** is the CT entry corresponding to the currently active compartment, and the **CCR.ID** is the ID of the currently active compartment. Second, the processor status register is augmented with a single bit that explicitly indicates whether the CPU is currently executing compartment code. We call this mode of operation *Compartment Mode*. The memory space layout, including the contents of the reserved memory, is shown in Figure 1.

3.3 Iso-X Operations and Instructions for Compartment Management

To support compartment management operations, several new instructions are added to the ISA and are directly supported by the Iso-X hardware. These instructions, along with the mode in which they can be accessed, are summarized in Table 2. The algorithmic descriptions of these instructions and their impact on the processor state are presented in [32].

comp_base	comp_size	page_count	comp_hash	cpt_base	cpt_size	flags
Points to the beginning of the compartment segment	Size of the compartment segment	Current number of pages mapped to the compartment	Used to perform compartment attestation	The starting physical page of CPT	Size of CPT	Describes compartment state

TABLE 1: Format of a CT Entry

Instruction	Arguments	Priv Mode	Comp Mode
① INIT	comp_id, comp_base, comp_size, cpt_base, cpt_size	Yes	No
② MAP	comp_id, virt_addr, phys_addr, page_permission_bits	Yes	No
③ ENTER	comp_id	No	No
④ ATTEST	None	No	Yes
⑤ REVOKE	comp_id, phys_addr	Yes	No
⑥ RESUME	comp_id	Yes	No

TABLE 2: Iso-X Instructions

3.4 Creating and Entering a Compartment

When a process requires the creation of a compartment, it passes the necessary information, such as the range of future compartment virtual memory pages, to the OS via a system call. Upon receiving this system call, the OS inspects its internal data structures to locate an unused compartment ID (*comp_id*) and finds the required number of contiguous free physical memory pages to hold the CPT for the compartment to be created. After that, the OS executes the new Iso-X instruction called INIT ①.

To execute the INIT instruction, the Iso-X hardware zeroes out the CT entry indexed by *comp_id*. The entry then is filled in based on the parameters of the INIT instruction. The *page_count* and *comp_hash* fields remain zeroed at this point. In addition, hardware clears the memory pages that will be used as CPT pages for this compartment. After this instruction completes execution, the empty compartment is initialized with no pages inside.

Populating the created compartment with memory pages is accomplished using another Iso-X instruction called MAP ②. This instruction adds the specified virtual-to-physical page mapping to the compartment's CPT with given permissions. Before making the page specified by the MAP instruction part of a new compartment, the Iso-X hardware checks the CMV bit of the corresponding physical page to ensure that this page does not already belong to another compartment since we do not allow double-mapping of the same physical page to different compartments. If the check passes, then the CMV bit is set, preventing further accesses to this page by untrusted code. The instruction also computes the hash of the entire page and extends the *comp_hash* field in the CT structure. To ensure the integrity of page mappings and permissions, both virtual page number and page permission bits are included as part of the page hash. Then, the *page_count* field of CT is incremented. Finally, the page's TLB entry is invalidated in order to prevent further accesses to that page from the UP.

To enter a compartment from an untrusted address space, the ENTER ③ instruction is used. The hardware sets up the CCR with the data corresponding to the *comp_id* that is used as an argument for this instruction. Once the CT entry has been loaded into CCR, the CPU starts executing the compartment code at a statically predefined location. The register state remains intact during this transition to allow the UP to pass data to the compartment.

3.5 Attesting Compartments and Building Trusted Channels

After a compartment is properly created, there are two important features required for the compartment to be useful to an external entity. First, the compartment needs to be able to prove to that external entity that it was properly loaded and is running on valid Iso-X hardware. Second, there must be a way for that external entity to create a secure communication channel with the compartment. Both of these issues are addressed simultaneously using the attestation mechanism of Iso-X.

Attestation in Iso-X takes the form of compartment certificates signed by the CPU. A compartment's certificate contains a hash of the compartment after loading as well as a copy of the compartment's unique public key. The CPU signs this certificate using a public/private key pair that is uniquely generated for the CPU at manufacturing time. This signed certificate can then be provided by a compartment to an external entity in order to prove that the compartment's integrity was not compromised during loading, it is running on valid Iso-X hardware, and to provide a copy of the compartment's public key to facilitate secure channel creation with the external entity.

When the loading of a compartment is completed, the attestation is performed using the ATTEST ④ instruction. When ATTEST is executed from within the compartment, the Iso-X hardware combines the *comp_hash* field of CCR.CT with the compartment's public key and signs the resulting data with the CPU's private key. The resulting certificate is then placed on the compartment metadata page. It can be sent by the compartment to outside entities for verification. After ATTEST is executed for the first time, the compartment is sealed. No additional code or non-empty data pages may be added to the compartment. Empty data pages may still be added in order to support dynamic growth of the stack and heap memory regions. To ensure the emptiness of such pages, pages are wiped before they are added to a compartment.

In order to securely communicate with trusted entities outside the compartment, the compartment leverages a secure communication library and standard cryptographic techniques to build a secure channel through the untrusted partition [48]. This functionality is similar to that used in HSM devices [58] to create trusted communication channels.

In order to make use of such channels without introducing a potential man-in-the-middle attack, the compartment must have its own, unique public/private key pair and securely communicate its public key to the other party in the communication. To this end, Iso-X allows compartments to execute some initialization code in isolated mode prior to compartment attestation. This allows the compartment to generate unique keys completely within the compartment. The initialization code then places the freshly generated public key at a specific location in the metadata page, allowing the hardware to use it during the attestation process. The private key is kept with the TP, ensuring that the OS cannot steal it.

To illustrate this attestation process, consider the following example. A programmer, Alice, writes a program containing both a trusted and untrusted partition. Within

the data portion of the TP, she includes a copy of her public key, PK_1. She also measures the correct hash of the trusted partition's code and data. She then sends her entire program to a remote machine containing an Iso-X processor, and her program is started. The TP of the program is loaded and protected, and right before calling `ATTEST` the TP generates a public/private key pair. It then places that new public key, PK_2 onto the compartment's metadata page. `ATTEST` is then executed, producing a CPU signed certificate containing the compartment's hash as well as PK_2. The compartment sends a copy of this certificate to Alice over the network, and Alice ensures that the hash is correct, hence verifying that the integrity of the compartment has not been compromised. Alice then creates a secure channel to the compartment. Alice uses PK_2 to verify that her channel is indeed with the compartment, and the compartment uses PK_1 to verify it is communicating with Alice.

3.6 Revoking Pages and Destroying Compartments

Revoking compartment pages and subsequently destroying compartments is accomplished via the `REVOKE` ⑤ instruction. The `REVOKE` instruction wipes off the page specified as its argument (thus preventing possible data leaks) and then clears the CMV bit corresponding to the page, allowing non-compartment code to access it. After a page has been revoked from the compartment, the *page_count* field of the CT entry corresponding to *comp_id* is decremented to reflect this change. If the compartment is left with no pages, it is considered destroyed. This instruction is used by the OS to reclaim compartment memory resources.

3.7 Dynamically Extending Compartments

In order to support heap and stack segments, Iso-X requires support for adding empty pages during regular compartment execution of an already attested and sealed compartment. However, allowing the OS to map an empty page could possibly compromise security. For example, the OS can first revoke a page and then map an empty page at the same virtual address. This allows the OS to zero out an arbitrary page within a compartment, thus opening a door for some attacks, such as non-control data attacks [18]. An example would be zeroing out a page that contains passwords. Our solution to this problem is to check the contents of a page when it is being revoked. If the page is empty, then it is allowed to be replaced later. If it is not empty, then the address of the revoked page is marked "revoked" in the CPT and no further pages can be mapped to that virtual address. This means that a compartment can explicitly allow pages (such as empty stack or heap pages) to be revoked by wiping them when they are no longer needed. If a non-empty page is revoked, however, then it cannot be replaced and the attack becomes impossible. This means the OS can still add empty pages at new virtual addresses, but it will no longer be capable of replacing arbitrary pages with empty ones.

3.8 Leaving Compartment Code

The Iso-X hardware performs additional actions when the program control flow transitions from an instruction belonging to a compartment to an instruction outside of it. Such a transition occurs in two cases: (1) to support an event that needs to be handled by the OS, such as a timing interrupt; and (2) to transition the control flow back to the untrusted partition after the compartment execution phase completes. We call this event `LEAVE` ⑦.

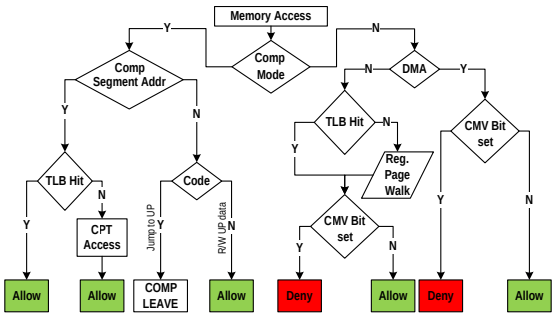


Fig. 2: Memory Access Data-flow in Iso-X

It is detected by the hardware during the instruction fetch stage when the CPU executes in compartment mode. When this event is detected, the current state of CCR.CT is saved in memory in the CT structure, all CPU registers are saved within the compartment metadata page and then they are wiped off. Finally, the processor exits the compartment mode.

To resume the compartment execution after it has been context switched, another Iso-X instruction – `RESUME` ⑥ is used. To implement it, the hardware restores the CCR.CT structure from the compartment's CT record. The compartment context is then restored from the compartment metadata page, and the CPU is switched to compartment mode.

3.9 Performing Memory Accesses in Iso-X

Securing memory accesses is at the center of Iso-X design. The integrity of the Iso-X system and compartment's data depends on the legitimacy of memory accesses. Therefore, all components responsible for performing memory accesses are trusted, including processor caches, the memory controller, memory buses and DRAM chips. (DMA devices are not trusted, as we will discuss in Section 3.10.) Memory protection in Iso-X is based on a modified paging mechanism. The Memory Management Unit (MMU) is the key component responsible for memory protection. The MMU checks the (trusted) TLB, and on a TLB miss it accesses the CPT. Disabling of the MMU by the OS/hypervisor is disallowed whenever active compartments exist in the system, otherwise those compartments could be compromised [35].

The memory data-flow in Iso-X is depicted in Figure 2. If the processor is executing in compartment mode, memory accesses are checked to see if they fall in the compartment segment range using existing segmentation support or similar hardware. If an access falls outside of the compartment segment, this situation is treated differently for data and instruction accesses. While outside code accesses generate the LEAVE event, data accesses are allowed, as it is the basis for the interaction of the compartment with the rest of the system. For an access within the compartment segment, the lookup proceeds using the TLB to determine the corresponding physical page number. A TLB hit occurs only if the *C* bit of the matching TLB entry is set. On a TLB miss, the CPT is accessed to get the translation, and the CMV bit of the corresponding physical page is checked. If it is not set, the access to a revoked page is detected and a security exception is raised.

If the processor is executing in regular (non-compartment) mode, then a TLB access is first performed to obtain the physical page number and the *C* bit. On a TLB hit (which occurs when the matching virtual page

entry is found and its *C* bit is set to zero), the memory access is allowed — it is a regular access outside of the compartment. Otherwise, on a TLB miss, a regular page walk of the conventional page tables is performed to obtain the translation, and then the CMV bit of the physical page is read from memory. If the CMV bit is zero, then the page mapping is installed in the TLB and the memory access is allowed to proceed. Otherwise, if the CMV value of the translated physical page is set, it signifies that the code outside of the compartment is attempting to perform a compartment access. Such an access is denied and a security exception is raised.

3.10 Direct Memory Accesses (DMA)

For systems that rely on the CPU to protect memory, DMA-enabled hardware is a security risk because it enables devices to access the physical memory directly, bypassing memory permission enforcement. Such attacks are known as DMA-attacks [59], [66]. Within the context of Iso-X, read and write DMA requests to compartment pages must be disallowed since they are controlled by possibly malicious I/O devices or system software. The specific implementation of a DMA request gating mechanism depends on the architecture of the I/O subsystem and buses. One way of controlling the DMA operations is by checking CMV bits in the Input-Output Memory Management Unit (IOMMU) prior to starting the DMA transfer.

Modern systems implement an MMU specifically for DMA transactions [5], [7] called IOMMU. The IOMMU intercepts DMA requests and performs translations from device addresses to a machine's physical addresses. Similar to a traditional MMU, the IOMMU uses page tables to perform address translation and also caches recently used page table entries in a special purpose TLB called IOTLB. To disallow DMA to compartment memory pages, the IOMMU hardware needs to be extended to check the CMV bit of the corresponding physical page before adding the mapping for this page to the IOTLB. This check needs to be performed during the I/O page walk. If the corresponding CMV bit is set, the IOTLB entry cannot be added, thus denying the DMA request. Novel IOMMU designs [8] allow to achieve very high IOTLB hit rates, thus making I/O page walks an infrequent event. The additional latency imposed by the CMV checks is a small fraction of the total DMA time, because DMA latency is dominated by the physical access to relatively slow I/O devices [14]. For example, according to [14], even the entire IOMMU has a negligible latency on bare-metal network throughput, and our design only adds a simple bit check to the IOMMU. Similarly, if we consider a disk access as the source of DMA operation, then a bit check is a negligible fraction of the disk access latency, which typically amounts to several milliseconds (tens of millions of the CPU cycles). This is true even when the CMV bits checked by IOMMU are not cached in the processor caches.

In cases when the requested DMA region spans multiple pages, the IOMMU hardware needs to check all corresponding CMV entries. In particular, when the IOMMU finishes translation of device addresses into a machine's physical addresses, it calculates the number of 4KB memory frames in the requested region. Next, it fetches the required number of bits from the CMV and OR-es individual values to obtain the final permission. If the OR-ing result is zero, then the IOTLB entry is set, otherwise the DMA operation is denied.

In addition to denying the establishment of the new IOTLB entries, Iso-X must restrict the usage of existing entries when new pages are added to compartments. To accomplish this, the corresponding IOTLB entries are invalidated when `MAP` instructions are executed. We note that existing IOMMUs already feature IOTLB entry invalidation capability [7]. In order to prevent the time-of-check time-of-use attack in which a page is added to a compartment during an in-flight DMA transaction, SGX relies on the mechanism used in regular IOMMUs. In particular, as described in [7], the CPU can issue a `COMPLETION_WAIT` command in order to synchronize the state of IOMMU tables with the rest of the system fabric.

3.11 Supporting Compartment Page Swapping

In terms of page swapping for the compartment pages, two approaches are possible. One solution is to simply pin the compartment pages to physical memory and disallow their swapping to the disk. With large DRAM capacities typically available on modern systems, this may not be a significant limitation. However, it is also possible to securely support compartment page swapping in Iso-X via two more ISA instructions (described below). Note that these instructions are optional and are only required if swapping support is necessary.

Before a compartment page can be swapped out, the Iso-X system must prepare it by measuring (hashing) and encrypting it. The OS is then allowed access to the page in order to swap it out. The confidentiality of the page is provided by the encryption, while its integrity is ensured by storing the page measurement in the internal Iso-X data structures. Storing the hash of the swapped out page ensures that when the page is swapped back in, the system can verify that it is indeed the same page. The Iso-X hardware prepares a page for swapping out using `SWAP_PREP` instruction, and returns it to the compartment later using the `SWAP_RET` instruction.

`SWAP_PREP` *(comp_id, virt_addr)*. First, the page to be swapped out is measured and the hash value is saved in the now unneeded CPT entry. To support such storage, each CPT entry in the system has to be wide enough to accommodate secure hash values. For example, SHA-2 produces 256-bit long hash values, so to store those, a 256-bit hash field is added to each CPT entry. For simplicity of implementation, this field is added next to the valid bit, permission bits and the physical frame number bits. Next, the page is encrypted with a symmetric key. This key is randomly generated by the hardware at boot time and stored in a register that is not accessible to software. Second, the valid bit in the CPT and the corresponding CMV bit are cleared. After these actions are completed, the page becomes available for read, write and DMA accesses by the OS. The OS can then initiate the DMA access to move the page to the disk. Finally, the TLB entry that covers the virtual page *virt_addr* is invalidated.

Swapping the page back into memory also requires co-operation of the OS and the Iso-X hardware. The OS is responsible for bringing the encrypted page from the disk back into memory, but further actions to return the page into the proper compartment can only be performed by the Iso-X hardware. The OS stores information about swapped out pages and is capable of distinguishing compartment pages from the regular ones. After a compartment page is moved to memory, the OS executes the `SWAP_RET` instruction to complete the swapping process as follows.

SWAP_RET *(comp_id, virt_addr, phys_addr)*. First, the CMV bit of the swapped in page is set, making the page accessible only by the Iso-X hardware. Next, the Iso-X hardware decrypts the page, measures it, and compares the resulting measurement with the one stored in the CPT entry that was saved during the swap-out. If the measurements match, the hardware replaces the CPT entry with a physical address of the page, provided by the OS. The valid bit in the CPT entry is then set, allowing compartment code to access the page. Otherwise, the swap in of such a page is disallowed. Finally, the old TLB entry needs to be invalidated to disallow accesses to that page from regular code.

4 Iso-X Security Analysis

In this section we will discuss a variety of potential attacks against Iso-X as well as how the system protects against them.

4.1 Directly Accessing Compartment Memory

An attacker may attempt to directly read or write compartment memory from outside the compartment.

4.1.1 Access from the OS.

The OS itself could attempt to directly access a compartment memory space by mapping the physical pages of the compartment into the page table for running OS execution context. From there, the OS could attempt to directly read or write the pages. Iso-X would prevent this attack when access is attempted. All compartment pages are marked as such in the compartment membership vector (CMV), and access to those pages is denied by the hardware if the CPU is not currently executing a compartment when the access occurs. This means that the OS, despite having full permissions over the page tables, would still be unable to access the contents of physical pages that have been assigned to a compartment.

4.1.2 Access from another compartment.

Another attack would involve setting up a malicious compartment controlled by an attacker and then using the OS to map pages from a victim compartment into it. (The pages would be double mapped.) This type of attack is not possible in Iso-X. The OS cannot directly modify the compartment page table (CPT) for any compartment, including a malicious, colluding compartment. Management of the CPT is handled directly by the hardware. An OS could add pages to its malicious compartment using the `MAP` instruction, however `MAP` prevents double mapping of compartment pages by ensuring that the page being added to a compartment is not already part of another compartment. A variation of this attack would be to first unmap a page from the victim compartment before mapping it into the malicious compartment (and hence there is no double mapping), but this attack is thwarted because the Iso-X hardware wipes pages when they are unmapped from a compartment.

4.1.3 Direct Memory Access (DMA).

An attacker could attempt to make use of DMA to cause the hardware to directly read or write the physical pages associated with a compartment. Under Iso-X, the pages involved in a DMA request are checked against CMV, and compartment pages are not permitted to be accessed using DMA.

4.1.4 Exploiting vulnerabilities in compartment code.

An attacker could analyze the compartment code and look for vulnerabilities such as buffer-overflows or pointer errors, that would allow an attacker to read or write memory directly, or perform a code reuse attack [62] to divert the control flow. Iso-X does not guarantee the correctness of compartment code, and as such does not explicitly address this type of attack. It is the responsibility of the compartment code developers to ensure that such vulnerabilities do not exist. For example, writing compartment code in a type-safe language will greatly minimize the surface for these types of attacks. In addition, techniques that explicitly protect from code reuse attacks can be deployed in conjunction with Iso-X [12], [26], [39], [51].

4.1.5 Hardware interrupts.

Iso-X compartments can be interrupted by standard interrupts, therefore an attacker could attempt to use interrupts to violate the control flow integrity of the compartment by modifying the contents of the PC register when it is pushed on the stack during an interrupt. Iso-X prevents this attack through careful handling of interrupts. If an interrupt occurs while a compartment is running, the hardware automatically saves the contents of all registers into the compartment's metadata page, wipes the registers, and then transfers control to the OS interrupt handling routine. When the OS later `RESUME`s the compartment, the registers are loaded from the metadata page and compartment execution continues. The metadata page is part of the sealed compartment, and hence inaccessible to the OS. This means that while the compartment is interrupted, the attacker is unable to modify the registers.

4.1.6 System management mode

Some processors implement a special execution mode called System Management Mode (SMM). In this mode, special purpose software (typically a part of firmware) is executed with an extremely high privileges and is able to access any physical memory in the system. SMM has been previously studied with regards to its ability to violate a system's security [29], [81]. SMM could also threatens Iso-X's integrity.

SMM in Iso-X can be handle in a few different ways. SMM could simply be modified to enable the memory controller to inspect the CMV bits for every memory operation that occurs. A preferable alternative, however, is to simply disable SMM whenever active compartments exist in the system.

4.1.7 Standard compartment leave/enter.

An attacker could attempt to manipulate the compartment registers after a standard `LEAVE` event. (Where a compartment voluntarily exits and transfers control to the untrusted partition.) However, this attack would fail for the same reason as the attack based on interrupts: the registers are stored on a memory page within the compartment.

4.2 Violating Compartment Integrity on Load

An apparently vulnerable time during the lifetime of a compartment is when it is being loaded. Prior to loading, the compartment contents may simply be residing on a disk, easily modified. The process of loading a compartment in Iso-X is described in detail in Section 3.5. In short, verifying correct loading of a compartment is

accomplished by the hardware generating a signed certificate containing a hash of the compartment contents. This certificate is verified by an external entity who knows the expected hash.

An attacker may attempt to compromise a compartment at load time using a variety of attacks.

4.2.1 Fail to load all pages.

A malicious OS could load the compartment, but intentionally not load all of its pages. While such an attack is possible under Iso-X, it would be detected by the external entity that verifies the compartment attestation. As a compartment is loaded and memory pages are added to it using the `MAP` instruction, a hash of the compartment is constantly updated with the contents of those pages, their permissions, and the virtual addresses associated with them. If an attacker does not load all the pages associated with a compartment, then the hash will not match and the external entity will detect the modifications as part of attestation.

4.2.2 Load additional pages.

Instead of failing to load all pages, a malicious OS could attempt to load additional pages into the compartment. This attack would fail for the same reason: the hash would be different, and the changes could be detected.

4.2.3 Misloading pages.

An attacker could load all of the compartment's pages, but load them into incorrect locations within the compartment. This attack would fail because the virtual address of each page is included in the hash along with its contents, binding contents and location with respect to the hash.

4.2.4 Manipulate the hash register directly.

An attacker could attempt to directly manipulate the hash value stored in the compartment table (CT), hence allowing the generation of an attestation certificate containing any desired hash. Under Iso-X, however, the hash value in the CT cannot be directly accessed by software under any circumstances, hence this attack is not feasible.

4.2.5 Forging the certificate.

When `ATTEST` instruction is executed, a certificate containing the compartment's hash is created and signed using the processor's unique private key. An attacker may attempt to forge this signature. Without access to the processor's private key, however, this type of attack is infeasible. In order to obtain the CPU private key, an attacker would need to have an insider present at the time of CPU manufacture, or would need to engage in the costly process of attempting to extract the key from the CPU. It is important to note that each Iso-X processor has its own, unique private key. If an attacker were to extract the key from a CPU, that key could be revoked by the manufacturer, and only that one processor would be affected.

5 Related Work

Several previous solutions for protecting systems from untrusted OS [52], [55], [80] rely on trusted software module. In these systems, the software TCB has to be formally verified to guarantee security. In addition, software approaches may incur substantial performance overhead, the extent of which depends on how often the critical security services and checks by the software TCB are needed. In the remainder of this section, we focus mostly on hardware-supported solutions with the exception of Inktag [38] and Virtual Ghost [22] as representatives of recent state-of-the-art software-only isolation schemes.

5.1 Hardware-Assisted Isolation

The Secret Protection (SP) architecture [30] supports a secure environment for executing trusted software modules that perform manipulations with secret keys. However, SP only supports one trusted software module per system. A more recent work, Bastion [17], supports many isolated compartments and is designed for modern software stacks supporting virtualization. However, Bastion relies on a modified hypervisor to be part of the TCB to provide some critical services. Similarly, SecureMe [20] uses a combination of memory cloaking (presenting the OS with encrypted view of memory), permission paging to provide a secure way for two applications to establish shared pages, and system call protection. It also relies on a small secure hypervisor for some of its tasks. Inktag [38] is a recent software-only solution that uses *paraverification*, a technique where the untrusted OS actions are monitored and verified by a trusted hypervisor, to provide isolation at the process granularity.

Other recent solutions proposed hardware support for protecting systems against attacks launched by a malicious hypervisor in virtualized systems. These include HyperWall [70], H-SVM [64] and HyperCoffer [77]. The granularity of isolation for these solutions is the Virtual Machine. The effort of [31] proposed an architecture based on non-inclusive memory permissions, thus not automatically giving a malicious OS or hypervisor access to the application code and data. The goal of these techniques is not to explicitly provide an isolated execution environment, but to protect memory pages from accesses by higher-privileged software.

XOM [45] proposed execute-only memory that allows instructions to be executed, but not manipulated in other ways. AEGIS [67], [69] provided a secure execution environment where any physical or software tampering becomes evident. Both AEGIS designs require that the application remains in secure mode throughout its execution. HyperWall [70] focuses on hypervisor-secure virtualization. It effectively removes the hypervisor from the TCB using hardware mechanisms. The CHERI capability-based architecture [76] offers some level of compartmentalization, however as the OS is responsible for saving and restoring capability state during context switches it is limited to protecting mutually untrusting domains from one another *within* a process. CODOMs [72] is an architecture that leverages the instruction pointer as a capability to provide isolation between protection domains. Mondrian memory protection [74] is an approach targeted at fine-grain memory protection. In this case, the permissions are stored in memory in a permissions table which is controlled by a privileged supervisor domain. Flicker [46] is a system that utilizes existing hardware to provide an isolated execution environment with a minimal TCB. This is accomplished through the use of the TPM [71] and CPU-based secure virtual machine extensions, e.g. Intel's Trusted Execution Technology (TXT) and AMD's Secure Virtual Machine (SVM) extensions. CrossOver [44] uses virtualization support available in commodity processors (specifically the `VMFUNC` instruction) to support efficient cross-world calls. The threat model considered in this work is significantly different than that of Iso-X and includes trusting the hypervisor for performing critical tasks.

IBM introduced an isolated execution design called SecureBlue++ [15]. SecureBlue++ is designed specifically for the PowerPC architecture. This mechanism cannot combine secure and insecure code in a single process. ARM's Trustzone is another commercial example - this architecture divides each layer of software into secure and insecure worlds and a trusted software monitor that controls the switching between the two worlds [6]. Intel's recent SGX/SGX2 security extensions is perhaps the most significant recent development in hardware-supported security in industry [9], [37], [47]. SGX is build around the concept of enclaves (hardware-enforceable containers) that provide isolated execution environment at the granularity that is determined by the application developers.

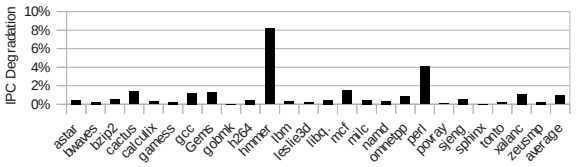
5.2 Comparing Iso-X with Intel's SGX

Since the SGX architecture shares similar goals with Iso-X, in this subsection we highlight the differences between the two approaches. Table 3 summarizes comparison of the two systems by key features specific to isolated execution environments. The main differences between the two designs are in the following areas:

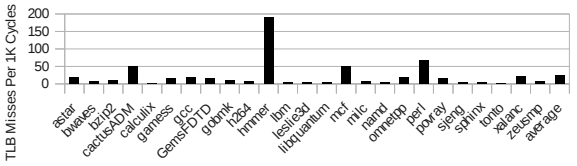
Compartment memory management and performance predictability: SGX requires all of the enclave's code and data to be physically located in a reserved memory region called the Enclave Page Cache (EPC). The EPC in SGX is a *fixed-size dedicated memory region*, which implies some limitations. For example, if the EPC cannot fit the memory pages for all enclaves, the OS would need to evict enclave pages often. Since encryption/decryption and integrity checks are required on every page eviction/return, this could slow down the system significantly, especially if the EPC size is sub-optimally configured at system boot (the EPC size cannot change dynamically during system operation). In contrast, Iso-X creates duplicate mappings in CPTs (Compartment Page Tables) and therefore allows compartment pages to be placed anywhere in memory. Only limited-size service data structures are stored in reserved memory and the memory overhead for each additional compartment is minimal.

A recently introduced next generation of SGX (called SGX2) still uses a fixed EPC size. The relevant difference between SGX2 and SGX is the ability to dynamically add empty pages to already created compartments — a capability that Iso-X provides by design. *However, the maximum size of EPC in DRAM is still statically determined in SGX2 at boot time and cannot be adjusted during execution. Consequently, while SGX/SGX2 directly protect all compartment pages within EPC, Iso-X allows the actual compartment pages to be located in the regular memory space, and controls access to them via CPTs (page mappings).*

Vulnerability to Page Fault-based Side-Channels: A recent work [78] demonstrated how a system running SGX can be vulnerable to a specific side-channel attack that leaks memory page access pattern to the malicious OS. In certain scenarios, such information is sufficient for recovering security-critical data, such as the encryption keys [63]. This attack is possible because SGX views the protected memory as a limited-size cache. In particular, the OS in SGX-based systems bears full responsibility for the management of enclave memory pages (except, of course, reading the content of these pages), including the capability to evict any page from the protected memory. When an enclave accesses one of its evicted pages, the OS is informed about such access. This makes the attack possible. The OS intentionally evicts some pages in order to detect later accesses to them.



(a) Iso-X Performance Degradation



(b) Combined TLB Miss Rates

Fig. 3: Performance results

In Iso-X, the number of protected memory pages is not limited. Therefore, the OS does not need to be given the capability to evict pages from protected to regular memory to free space in the protected memory. This makes Iso-X immune to such side-channel attacks when optional swapping is disallowed (compartment pages always reside in memory). In particular, an Iso-X system can be configured in a way that restricts the OS from executing the swapping instructions. Moreover, it is possible to disable such functionality for some compartments, while keeping it for others. Whether or not swapping is enable can be configured in a compartment's certificate, making the compartment's owner (the user) aware if the compartment is vulnerable to page fault-based side-channel attacks.

6 Performance Evaluation

We evaluate performance overhead due to extra memory permission checks and also the overhead of one-time or infrequent events such as compartment creation, destruction, and page swapping.

6.1 Permission Access Overhead

With every memory access, Iso-X must check the CMV bits. This is the only ongoing overhead that Iso-X adds during steady-state execution. To model this impact, we simulated the SPEC2006 CPU benchmarks [65] using MARSSx86 full system x86-64 simulator [2]. The simulated processor configuration is depicted in Table 4. For each benchmark, we simulated 5 billion committed instructions. We assume that the entire benchmark code is executed inside a compartment.

Since the CMV bits are stored in the processor caches, our model captures this impact. We expect the performance loss to be small for two reasons. First, since the CMV bits are also stored in the processor TLBs (in the form of `C` bits as described in Section 3), the memory accesses to retrieve them are only performed on TLB misses. Second, a single cache line containing CMV bits covers many adjacent pages, therefore a high cache hit rate is expected.

Figure 3a shows the decrease in the commit IPCs (Instructions per Cycle) of SPEC2006 benchmarks for an Iso-X system normalized to a baseline system without Iso-X. As seen from the results, Iso-X performance loss compared to the baseline architecture is below 1% on the average in both secure and non-secure modes. The largest loss among individual benchmarks was observed for *hmmer* — about 8%. Figure 3b depicts the combined

	Granularity of protection	System software in TCB	Limited isolated execution environment	Hardware Attestation mechanism	Dynamic protected space	Requires encrypted executable	Secrets can reside anywhere	Memory and memory bus are trusted
Iso-X	Virtual memory region	N	N	Y	Y	N	Y	Y
SGX [47]	Virtual memory region	N	N	Y	N	N	N	N

TABLE 3: Comparison of Iso-X with SGX

Parameter	Configuration
Datapath	4-way superscalar, 128-entry ROB, 64-entry Issue Queue, 96-entry LSQ
Inst. & Data TLBs	64-entry, Fully Associative
L1 I & D Caches	32 KB, 8-way, 64B line, 2 cycles
L2 Unified Cache	256KB, 8-way, 64B line, 10 cycles
L3 Unified Cache	8MB, 16-way, 64B line, 40 cycles
Memory latency	150 cycles

TABLE 4: Configuration of the Simulated x86-64 Processor

miss rates for both data and instruction TLBs for the Iso-X system.

The cache miss rates specific to the metadata accesses were extremely low for Iso-X (about 1.3% on average). In addition, the small metadata size causes negligible pollution to the cache which results in little to no increase in miss rate for normal data.

In order to provide an estimate of the area overhead and the impact on the cycle time, we implemented Iso-X permission accesses on the OpenRISC processor core using an Altera DE0 Nano FPGA board. We used OpenRISC version 3.1 [42] and Altera Quartus II 13.1 for our timing, area and power analysis. The OpenRISC processor is a 32-bit in-order pipelined architecture with 16KB data and instruction caches, 32 registers and 64-entry separate data and instruction TLBs. In order to estimate the runtime overhead of Iso-X, we implemented the *C* bit checks on every memory access, as well as additional memory reads from CMV to refill the corresponding *C* bit on TLB misses. Since all Iso-X violations are treated as high priority exceptions, the routing of the Iso-X exception signal is on the critical path of the processor, resulting in a slight frequency decrease. The checks reduced the maximum frequency of the processor only by 2%. However, in a commercial design with ASIC tools, this extra delay can be optimized to avoid an increase in cycle time. The CMV bits for all of the system's pages occupy only 512 Bytes of memory for this implementation, since there is only 32MB of physical memory and OpenRISC uses 8KB pages. The effect of the dynamic runtime Iso-X logic on the core area is only 0.65% and it has a 1% increase in dynamic power. In an ASIC implementation of Iso-X with out-of-order processor, these overheads will be even lower, as the out-of-order structures will contribute to a larger fraction of the chip area and power.

6.2 Overhead of Compartment Operations

We now evaluate other overheads of Iso-X, primarily those involved in the creation and destruction of compartments. These results are summarized in Table 5. This table shows the number of cycles that each compartment operation takes. In addition, each operation is broken down into actions that it requires and the associated costs. The figures in the table were obtained by running a suite of micro-benchmarks, which we developed on an Intel Core i7-4700MQ CPU running at a frequency of 2.4GHz.

The population of compartment memory depends on the number of pages that must be added to the compartment, and destruction depends on the number of pages

Operation	Actions(s)	Cost (Cycles)
Create	System Call	138
	Find Free Enc	2,107
	`INIT`	2
	Total	2,247
Populate	Hash Page(s)	6.172M
Attest	Sign Hash	1.241M
Revoke Page, Add Empty Page	Zero Page	596
Destroy	Revoke All Pages	52,379
Interrupt	`LEAVE`	52
Resume	`RESUME`	26
Swap Page Out	Hash Page	70,357
	Encrypt Page	32,744
	Total	103,101
Swap Page In	Decrypt Page	22,593
	Hash Page	70,357
	Total	92,950

TABLE 5: Compartment Operation Overheads

that must be removed. For this example, we have used the sizes of `sshd`, which requires 89 4KB pages (88 for the program itself and a single CPT page), to calculate the total costs. To compute the total cost of compartment destruction, we assumed that all 88 pages need to be revoked. Furthermore, the frequency of some operations (i.e. revoke, interrupt, resume, swap out, and swap in) will vary since they depend on the overall system load. The numbers reported for these operations represent the cost of each invocation. We evaluated the following cryptographic functions: SHA-256 for hashing, 1024-bit RSA for certificate signing, and 128-bit AES-CBC for encryption. We used the *polarssl* library for hashing, signing, and encryption. This means that the costs of hashing and signing are representative of a hardware implementation which uses the regular CPU datapath to perform these operations, rather than dedicated hardware. Note that the polarssl implementation of AES uses Intel's AES-NI instructions.

Out of the operations shown in Table 5, the most expensive ones are hashing and signing — each taking more than one million cycles for a compartment of the considered size. However, these operations only occur once during the lifetime of a compartment. Therefore, the overhead of these operations can be tolerated. All other overheads presented in Table 5 are much smaller.

Some of the crypto operations can be substantially accelerated by deploying dedicated crypto-engines. For example, the SHA-256 hashing on a dedicated crypto-engine clocked at 170MHz requires 0.125 cycles/byte according to [49], which is about 10x faster than the software implementation reported in Table 5. As another example, AES encryption performed on a separate engine clocked at 340MHz requires 0.69 cycles/byte [36], which is about 1.6x faster than encryption that uses Intel's AES-NI instructions. Note that with a crypto-engine, the encryption can be done in the background, freeing up the CPU core to continue execution.

In summary, these estimations demonstrate that regardless of how the encryption/hashing/attestation logic is implemented (either within the main CPU or through an accelerator), the performance overhead of these activities is tolerable given the infrequent nature of these operations.

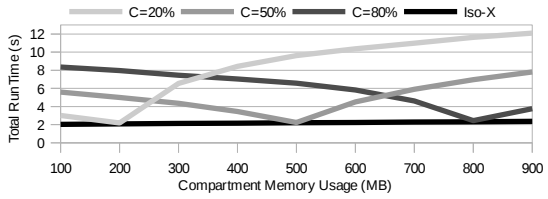


Fig. 4: Effect of Static Memory Provisioning

6.3 Impact of dynamic memory reservation

One of the key differences between Iso-X and SGX is that SGX requires that compartments reside in a memory region to be statically reserved during boot time. There are a number of situations where the decision of how much memory to allocate to compartments at boot time could be difficult. For example, this can be a problem on memory-constrained systems such as embedded devices or smart phones, or on systems with drastically changing workloads such as cloud servers. In contrast, the Iso-X system can dynamically protect any physical page in the system.

To provide a basic comparison between this difference in the models of SGX and Iso-X, we show the effect of a statically provisioned compartment memory region on a system where the overall memory pressure is high. In particular, we created a virtual machine with a total of 1000MB of RAM and the host file system cache disabled (i.e. VM disk writes go directly to an SSD drive). The size was selected to be representative of modern smart phones such as the iPhone 6. Inside this VM we ran a stripped down copy of 32-bit Debian Linux. We partitioned the memory inside the VM to represent compartment and untrusted partitions. Inside each partition we run a benchmark that allocates and accesses memory of a configurable size. By changing this size, we can shift the pressure from the trusted to the untrusted partition. On a page fault, we encrypted and sealed the pages on the compartment partition consistent with swapping support in Iso-X.

We fix the overall memory demand, but vary the percentage of workload pages that need to be isolated. Because the memory pressure is high, only a correctly provisioned compartment memory size allows the system to function without thrashing. If the compartment memory size is under-provisioned (the left hand side of Figure 4) many page faults occur in that memory region, incurring expensive compartment side swaps (which require encryption and hashing). Alternatively, if the compartment side is over-provisioned, the untrusted partition is under-provisioned and page faults occur in that region. Meanwhile, Iso-X is able to dynamically grow each partition to the size it needs and avoids thrashing on either side. Note that if the system uses magnetic drives instead of SSD, the impact of incorrect provisioning will be substantially higher.

7 Software Support for Iso-X

In this section, we describe software modifications needed to allow applications to use the Iso-X system.

7.1 System Software Support

Iso-X requires modest modifications to the OS code. In particular, the OS maintains a list of compartments created by each application and provides a system call interface to user programs for creation and management of compartments. Additionally, the system software is responsible for allocating contiguous physical memory pages for constructing the CPTs during the process of compartment initialization. The OS needs to be aware of which pages belong to compartments, as those pages are not accessible for normal page allocations. To support physical page reclamation from compartments, the system software also needs to be modified to perform this service through the use of the `REVOKE` instruction. With these modifications, the OS controls the allocation of compartments and their resources, but has no access to compartment memory.

7.2 Preparation and Initialization of Iso-X Programs

Applications designed to run on the Iso-X system must be compiled in a particular way. Specifically, Iso-X requires all of its pages to reside in contiguous memory segments, forming a larger compartment segment inside the virtual address space of the untrusted partition. Executables must preserve such continuity of the segments. In most cases, this requirement does not translate into any significant complications at the level of source code. This is because placing compartment-associated code and data objects into a dedicated compartment segment can simply be achieved using a combination of compiler attributes and modified linker scripts. The Iso-X instructions can be inserted as embedded assembly without requiring any changes to the compiler. If compartment code needs to include any library functions (such as the dynamically linked libc functions) while in compartment mode, these functions are copied into the compartment code segment and are statically linked. Otherwise, any attempt to execute library functions will result in leaving the compartment.

Compartment sections containing dynamic data, such as the heap and the stack, are initialized by the untrusted partition at runtime from its own heap memory. The layout of a compartment's virtual address space is depicted in Figure 1b.

Executables containing a compartment segment are executed in a regular fashion. When the application needs to initialize a compartment, it initiates and loads all compartment pages into memory, including all of the empty pages. After that, it prepares a data structure containing relevant information about the compartment, including the compartment segment base address and the size of the compartment segment. Next, a system call is made which passes a pointer to this data structure to the OS as an argument. In response, the OS finds a free compartment ID and executes the `INIT` instruction. After the instruction completes, the OS checks all memory pages in the compartment segment, obtains their physical addresses, and executes a sequence of `MAP` instructions to map all of the compartment's initial pages. At this point the compartment initialization process completes. The OS then returns to the untrusted partition, which uses the `ENTER` instruction to enter the newly created compartment.

7.3 Iso-X Application Structure

To provide strict isolation between compartments and the untrusted partition, the Iso-X developers can leverage code analysis tools, such as static information flow analysis [27], to verify that the isolation property is fully achieved. Porting existing applications to Iso-X appears more challenging due to a large number of cross-references between different parts of the code. For this case, methods of privilege separation [40], including automated program partitioning [16] can be utilized.

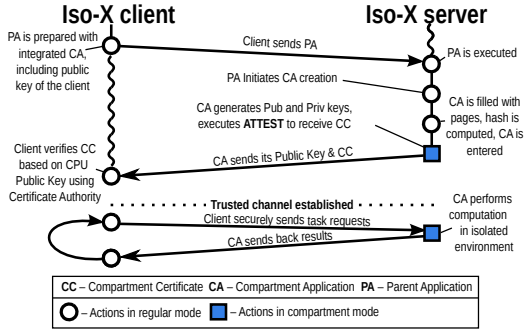


Fig. 5: Usage Flow in Cloud Scenario

8 Iso-X Application Examples

Some classes of applications can directly benefit from the Iso-X system by storing their secrets and performing computations on sensitive data inside the compartments. The examples include banking and e-commerce applications, digital rights management applications, password managers, and disk encryption software. It is also interesting to deploy hardware compartments for stronger guarantees in support of software isolation mechanisms (e.g., Java Isolation [11]). In addition to these, we describe two more scenarios where Iso-X can provide benefits.

8.1 Isolated Execution in A Remote Cloud

Iso-X can significantly improve security of cloud computing. In all currently commercially available clouds, there is a layer of supervisor software with unrestricted permissions that is capable of compromising user's private data. Therefore, prospective users of the cloud have to accept the risks that their sensitive data can be leaked.

In contrast, computational clouds augmented with Iso-X functionality can provide their users with guarantees that programs that handle sensitive data will always be executed in a hardware-protected isolated environment, as described in [13] and [60]. Figure 5 depicts an example of actions that need to be taken by the user of the cloud service (the Iso-X Client) and the provider of the computational cloud resources (the Iso-X Server) to guarantee secure execution. The Iso-X client first initiates remote compartment creation, loads the compartment with the desired code, establishes a trusted channel with the compartment, and attests and verifies the integrity of the channel. After the trusted channel with the attested compartment is set up, the Iso-X client sends out computational tasks to the compartment for secure execution. The results of these tasks are then returned to the client using the same trusted channel.

This model is similar in principle to the design of Hardware Security Modules (HSM) [58]. The Iso-X system provides functionality that is similar to the recently introduced CloudHSM [1] module from Amazon, but without the need for a separate hardware device.

8.2 Secure Machine Attestation

Code in compartments can be used to periodically perform machine attestation to ensure that the platform has not been tampered with. More specifically, the essential components of the anti-virus and anti-malware software can be placed inside a compartment, which makes their code bases tamper-resistant. Even higher security can be achieved by periodic attestation of these compartments using the Iso-X attestation mechanism. A recent example of an approach that is close to this philosophy is the McAfee/Intel Deepsafe technology [3], where some parts of user-level anti-virus programs are made tamper resistant using hardware support.

8.3 Partially Isolated Applications

Although the main goal of the Iso-X system is to provide a hardware framework for building fully-isolated applications, there can be some benefit from running programs that feature only partial isolation. Most current privilege separation solutions split a program into two parts, depending on whether the code can be tampered with by the user input or not. These two parts are executed as individual processes and use some form of inter-process communication (IPC) to interact, rather than simple function calls. There are significant performance implications to this approach due to an increased number of system calls, context switches, and IPC operations. Iso-X can be used to mitigate such costs. Instead of relying on expensive IPC mechanisms, Iso-X relies on switches between compartment and regular (non-compartment) mode, achieving very low-overhead communication. For example, a simple remote-procedure-call that does nothing but return, takes about 16.6K cycles when both client and server are running on the same machine. In contrast, the cost of entering and leaving a compartment is only 78 cycles. Both numbers were obtained on an Intel Core i7 4700MQ CPU clocked at 2.4GHz.

9 Concluding Remarks

Providing trusted and isolated execution environment for secure execution in the presence of potentially compromised system software layers is a challenging task. In this paper, we introduced Iso-X — a hardware-assisted framework for isolated execution. Iso-X requires modest hardware support: six new ISA instructions, secure compartment page tables and associated logic, a bitmap storing the identity of compartment memory pages, and a few registers. These mechanisms allow the code executed inside a compartment to be protected from the OS/hypervisor, from other code executed in the untrusted domain, from DMA operations, and also from other compartments. In addition, the Iso-X trusted computing base (TCB) only includes the software located inside the compartment and the Iso-X hardware itself. Iso-X offers advantages over existing hardware-based isolation proposals in the granularity of protection and memory allocation flexibility. Moreover, we demonstrated that the security benefits of Iso-X are achieved with negligible overhead. We prototyped critical components of Iso-X integrated with an OpenRISC core and evaluated them on an FPGA. Furthermore, we showed that the performance overhead of compartment creation, modification and destruction are tolerable especially given that these actions are not required frequently.

10 Acknowledgement

This publication was made possible by the support of the NPRP grant 4-1593-1-260 from the Qatar National Research Fund. The statements made herein are solely the responsibility of the authors.

References

[1] "AWS CloudHSM," 2013, http://aws.amazon.com/cloudhsm. Retrieved August 2013.
[2] "MARSSx86: Micro-ARchitectural and System Simulator for x86-based Systems," 2013, http://marss86.org. Simulator source code and documentation.
[3] "Root out rootkits: An inside look at mcafee deep defender," 2013.
[4] "Intel Software Guard Extensions Programming Reference,"

2014.
[5] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel Virtualization Technology for Directed I/O." *Intel technology journal*, vol. 10, no. 3, 2006.
[6] T. Alves and D. Felton, "ARM TrustZone: Integrated Hardware and Software Security," in *Information Quarterly*, 2004.
[7] *AMD I/O Virtualization Technology (IOMMU) Specification*, AMD, Feb. 2015, rev. 2.62.
[8] N. Amit, M. Ben-Yehuda, and B.-A. Yassour, "IOMMU: Strategies for mitigating the IOTLB bottleneck," in *International Symposium on Computer Architecture*. Springer, 2010, pp. 256–274.
[9] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.
[10] Anonymous, "Xbox 360 Hypervisor Privilege Escalation Vulnerability," 2007, available online: http://www.securityfocus.com/archive/1/461489.
[11] G. Back, W. C. Hsieh, and J. Lepreau, "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java," in *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation (OSDI)*, 2000.
[12] M. Backes, M. Planck, and S. Numberger, "Oxymoron: making fine-grain memory randomization practical by allowing code sharing," in *USENIX Security Symposium (USENIX Security)*, 2014.
[13] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.
[14] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. Van Doorn, "The price of safety: Evaluating IOMMU performance," in *The Ottawa Linux Symposium*, 2007, pp. 9–20.
[15] R. Boivie and P. Williams, "SecureBlue++: CPU Support for Secure Executables," 2013.
[16] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004, pp. 57–72.
[17] D. Champagne and R. Lee, "Scalable architectural support for trusted software," in *Proceedings of HPCA*, 2010.
[18] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of the USENIX Security Symposium*, 2005, pp. 177–192.
[19] X. Chen, T. Garfinkel, E. Lewis, P. Subrahmanyam, D. Boneh, J. D. Dan, and R. Ports, "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of ASPLOS*, 2008.
[20] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "Secureme: A hardware-software approach to full system security," in *Proc. International Conference on Supercomputing (ICS)*, Jun. 2011.
[21] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 20 16. http://eprint. iacr. org, Tech. Rep.
[22] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proc. ASPLOS*, 2014.
[23] "CVE-2007-4993: Xen guest root can escape to domain 0 through pygrub," 2007.
[24] "CVE-2007-5497: Vulnerability in XenServer could result in privilege escalation and arbitrary code execution," 2007, available online:http://support.citrix.com/article/CTX118766.
[25] "CVE-2008-2100: VMware Buffer Overflows in VIX API Let Local Users Execute Arbitrary Code in Host OS," 2008.
[26] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: a detection tool to defend against return-oriented programming attacks," in *Proceedings of ASIACCS*. ACM, 2011, pp. 40–51.
[27] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.
[28] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: A low-complexity mitigation of cache side-channel attacks," in *ACM Transactions on Architecture and Code Optimization*, Jun. 2012.
[29] L. Duflot, D. Etiemble, and O. Grumelard, "Using CPU system management mode to circumvent operating system security functions," *CanSecWest/core06*, 2006.
[30] J. Dwoskin and R. Lee, "Hardware-rooted trust for secure key management and transient trust," in *Proceedings of CCS*, 2007.
[31] J. Elwell, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "A non-inclusive memory permissions architecture for protection against cross-layer attacks," in *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2014.
[32] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A flexible architecture for hardware-managed isolated execution," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 190–202.
[33] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Covert channels through branch predictors: a feasibility study," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2015, p. 5.
[34] ——, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 10, 2016.
[35] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014.
[36] A. Hodjat, D. D. Hwang, B. Lai, K. Tiri, and I. Verbauwhede, "A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18-μm CMOS technology," in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*. ACM, 2005, pp. 60–63.
[37] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, and V. Phegade, "Using innovative instructions to create trustworthy software solutions," in *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.
[38] O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel, "InkTag: Secure Applications on an Untrusted Operating System," in *Proceedings of ASPLOS*, 2013.
[39] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low overhead mitigation of code reuse attacks," in *Proceedings of ISCA*, 2012.
[40] D. Kilpatrick, "Privman: A library for partitioning applications." in *USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 273–284.
[41] K. Kortchinsky, "Hacking 3D (and Breaking out of VMWare)," in *BlackHat USA*, 2009.
[42] D. Lampret, C.-M. Chen, M. Mlinar, J. Rydberg, M. Ziv-Av, C. Ziomkowski, G. McGary, B. Gardner, R. Mathur, and M. Bolado, "Openrisc 1000 architecture manual," *Description of assembler mnemonics and other for OR1200*, 2003.
[43] R. B. Lee, P. C. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for protecting critical secrets in microprocessors," in *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*. IEEE, 2005, pp. 2–13.
[44] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan, "Reducing world switches in virtualized environment with flexible cross-world calls," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 375–387.
[45] D. Lie, M. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proceedings of ASPLOS*, 2000.
[46] J. McCune, B. Parno, A. Perrig, M. Reiter, and A. Seshadri, "How Low Can you Go? Recommendations for Hardware-Supported Minimal TCB Code Execution," in *Proc. ACM International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
[47] F. McKeen, I. Alexandrovich, A. Berenzon, C.Rozas, H. Shafi, V. Shanbhogue, and U. Svagaonkar, "Innovative instructions and software model for isolated execution," in *Wkshp. on Hardware and Architectural Support for Security and Privacy, with ISCA'13*, 2013.
[48] R. Merkle, "Secure communications over insecure channels," *Communications of the ACM*, vol. 21, no. 4, pp. 294–299, Apr. 1978.
[49] H. E. Michail, G. S. Athanasiou, V. Kelefouras, G. Theodoridis, and C. E. Goutis, "On the exploitation of a high-throughput sha-256 fpga design for hmac," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 5, no. 1, p. 2, 2012.
[50] "NIST National Vulnerability Database," 2012, available online at http://nvd.nist.gov.
[51] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "Gfree: Defeating return-oriented programming through gadget-less binaries," in *Proceedings of ACSAC*, 2010, pp. 49–58.
[52] K. Onarlioglu, C. Mulliner, W. Robertson, and E. Kirda, "Privexec: Private execution as an operating system service," in *IEEE Symposium on Security and Privacy*, May 2013.
[53] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vadudevan, "OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms," in *Proceedings of CCS*, 2013.
[54] D. Perez-Botero, J. Szefer, and R. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *Proceedings of the Workshop on Security in Cloud Computing (SCC)*, 2013.
[55] R. Riley, X. Jiang, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," in *Recent Advances in Intrusion Detection (RAID)*, 2008, pp. 1–20.
[56] J. Rutkowska, "Introducing the Blue Pill," 2006, available Online: http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[57] ——, "Thoughts on intels upcoming software guard extensions," *Invisible Things Lab*, 2013.
[58] R. Sanchez-Reillo, C. Sanchez-Avila, C. Lopez-Ongil, and L. Entrena-Arrontes, "Improving security in information technology using cryptographic hardware modules," in *Security Technology, 2002. Proceedings. 36th Annual 2002 International Carnahan Conference on*. IEEE, 2002, pp. 120–123.
[59] F. L. Sang, V. Nicomette, and Y. Deswarte, "I/O attacks in Intel PC-based architectures and countermeasures," in *SysSec Workshop (SysSec), 2011 First*. IEEE, 2011, pp. 19–26.
[60] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 38–54.
[61] "CVE Details: The Ultimate Security Vulnerability Datasource," 2013, accessed Nov. 2013 at http://cvedetails.com.
[62] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of CCS 2007*, S. De Capitani di Vimercati and P. Syverson, Eds. ACM Press, Oct. 2007, pp. 552–61.
[63] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets: Defenses against pigeonhole attacks," *arXiv preprint arXiv:1506.04832*, 2015.
[64] S.Jin, J.Ahn, S.Cha, and J.Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *Proceedings of MICRO*, 2011.
[65] C. D. Spradling, "SPEC CPU2006 benchmark tools," *SIGARCH Comput. Archit. News*, vol. 35, no. 1, pp. 130–134, 2007.
[66] P. Stewin and I. Bystrov, "Understanding DMA malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2012, pp. 21–41.
[67] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proceedings of ICS*, 2003.
[68] ——, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of MICRO*, 2003.
[69] G. Suh, C. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of ISCA*, 2003.
[70] J. Szefer and R. Lee, "Architectural support for hypervisor-secure virtualization," in *Proceedings of ASPLOS*, 2012.
[71] "TPM 1.2 Main Specification," 2011, available online at: http://www.trustedcomputinggroup.org/tpm-main-specification/.
[72] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: protecting software with code-centric memory domains," in *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 2014, pp. 469–480.
[73] Z. Wang and R. Lee, "A novel cache architecture with enhanced performance and security," in *Proc. International Symposium on Microarchitecture (MICRO)*, Dec. 2008.
[74] E. Witchel, J. Cates, and K. Asanovic, "Mondrian memory protection," in *Proceedings of ASPLOS*, 2002.
[75] R. Wojtczuk, "Subverting the Xen hypervisor," in *BlackHat USA*, 2008.
[76] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 2014, pp. 457–468.
[77] Y. Xia, Y. Lin, and H. Chen, "Architecture Support for Guest-Transparent VM Protection from Untrusted Hypervisor and Physical Attacks," in *Proceedings of HPCA*, 2013.
[78] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," 2015.
[79] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, L3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 719–732.
[80] F. Zhang, J. Chen, H. Chen, and B.Zang, "Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization," in *Proceedings of SOSP*, 2011.
[81] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.
[82] D. Zovi, "Hardware Virtualization Based Rootkits," in *Black-Hat USA, 2006*, 2006, available Online: http://blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf.

Dmitry Evtyushkin Is a PhD student in the Department of Computer Science at SUNY Binghamton. His research interests are in the areas of computer architecture and secure system design, with specific focus on architectures for isolated execution and investigation of covert channels and side channels through shared processor resources. He received his undergraduate degree from Moscow Institute of Electronics and Mathematics in 2011.

Jesse Elwell recently graduated with a PhD in Computer Science from SUNY Binghamton and has accepted a position at Vencore Labs as a research scientist. His research interests are in the areas of computer architecture, security, and system software.

Meltem Ozsoy is a Research Scientist at Intel Labs, Hillsboro, OR. She received her PhD in Computer Science from SUNY Binghamton in 2015. Her research interests are in the areas of computer architecture and secure system design.

Dmitry Ponomarev is a Professor in the Department of Computer Science at SUNY Binghamton. His research interests are in the areas of computer architecture, secure system design, power-aware systems and high-performance computing. He received his PhD in Computer Science from SUNY Binghamton in 2003.

Nael Abu-Ghazaleh is a Professor in the Department of Computer Science and Engineering and the Department of Electrical and Computer Engineering at the University of California at Riverside. His research interests are in the areas of secure system design, parallel discrete event simulation, networking and mobile computing. He received his PhD from the University of Cincinnati in 1997.

Ryan Riley is an Assistant Professor at Qatar University. His research interests are in the area of computer security. He received his PhD from Purdue University in 2009.