Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution

Dmitry Evtyushkin*, Jesse Elwell*, Meltem Ozsoy*, Dmitry Ponomarev*, Nael Abu Ghazaleh† and Ryan Riley‡
*State University of New York at Binghamton, †University of California at Riverside, ‡Qatar University
{devtyushkin,jelwell,mozsoy,dima}@cs.binghamton.edu, nael@cs.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 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. Iso-X offers higher memory flexibility than the recently proposed SGX design from Intel, allowing both fluid partitioning of the available memory space and dynamic growth of compartments. An FPGA implementation of Iso-X runtime mechanisms shows a negligible impact on the processor cycle time.**

***Keywords*-hardware security; isolated execution;**

I. 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 system software stacks include hypervisors to support virtualization and one or more guest operating systems (OSs) running on top of them. OS and virtualization layers are growing into large and very complex pieces of code. Indeed, modern OS kernels have tens of millions of lines of code [23], and it is virtually impossible to design them without exploitable vulnerabilities. For example, 189 new vulnerabilities were reported for the Linux kernel in 2013 [41], of which 6 lead to arbitrary code execution, 13 to memory corruption and 26 to privilege escalation, for example by using *return-to-user* attacks [22].

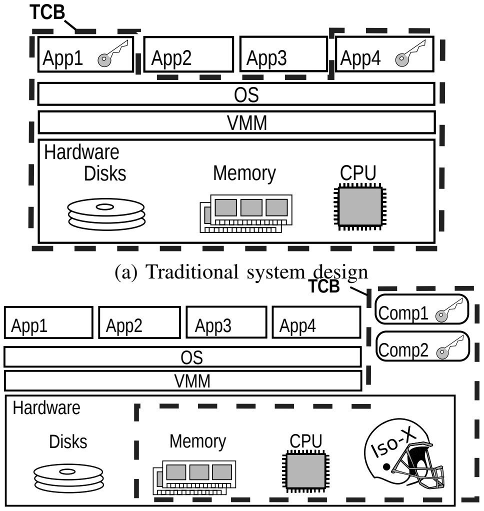
To exacerbate the situation, modern hypervisors (such as Xen or KVM) are also rapidly becoming large pieces of software with hundreds of thousands of lines of code, and many vulnerabilities are discovered in them every year. A recent study [35] analyzed and classified hypervisor vulnerabilities and attack surfaces. According to the study, 59 vulnerabilities have been identified in Xen and 38 in KVM. Many recent attacks on hypervisors, exploiting these vulnerabilities, have been successfully demonstrated [5], [13], [14], [15], [24], [38], [50], [54].

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 compartments* [4], [7], [8], [10], [11], [20], [21], [26], [27], [30], [44]. These compartments are inaccessible to the system software layers and are managed either entirely by the hardware [7], [27], [30], [34] or by a special layer of secure software that is sometimes assisted by hardware [8], [10], [11], [21], [26], [44]. The idea of supporting secure isolated environments has also received considerable attention from industry. Amazon recently announced its CloudHSM service [1] to support isolation and provide secure execution in the cloud environment. Furthermore, Intel introduced SGX extensions to x86 processors that are built around the concept of secure enclaves (compartments) [4], [20], [30].

In this paper, we propose Iso-X (Isolated eXecution) - a hardware-managed framework for supporting a fine-grained and flexible 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 provable 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 compartmentalization 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. We demonstrate that the performance overhead of Iso-X is negligible and predictable for both secure and non-secure mode execution. In addition, we show that the overhead of creating and destroying compartments is also tolerable, considering that these operations are not required often.

We evaluate and validate the Iso-X design in a number of ways. First, we develop a fully-functional software environment for simulating Iso-X within the OpenRISC simulator [25]. In addition, we integrate the Iso-X runtime



(a) Traditional system design

(b) Iso-X system design

Figure 1: TCB Comparison of the Traditional System and the Iso-X System

mechanisms into the HDL description of the OpenRISC processor core. These two components show that the critical mechanisms for Iso-X are functional and estimate their complexity in an FPGA context. Next, we measured the performance overhead of Iso-X on a full-system x86 simulator using the SPEC 2006 benchmarks to provide an estimate of performance impact on typical systems and benchmarks. Finally, we estimate the performance overhead of one-time, infrequent or optional Iso-X mechanisms (such as initialization, crypto-operations and swapping) using micro-benchmarking on real hardware.

In summary, this paper makes the following contributions:

- We propose and 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.
- We present evaluation of an integrated HDL implementation of the runtime Iso-X hardware with an OpenRISC processor core [25]. 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 provide a detailed comparison of Iso-X with Intel's recently announced SGX architecture and also place the proposed scheme into the context of other related efforts from academia.

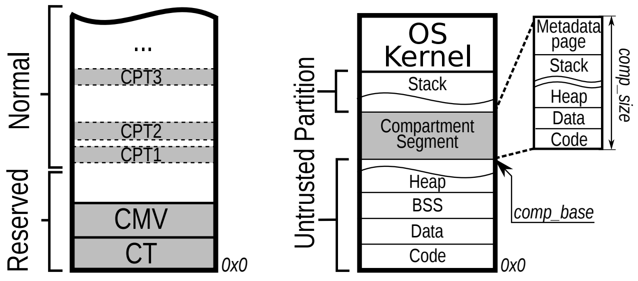
II. Threat Model and Assumptions

We assume that any portion of the system software stack, including the OS and the hypervisor, can be potentially compromised. Only software that runs in a special hardware-supported compartment mode can be fully trusted. The software trusted computing base (TCB) in the Iso-X system is thus limited to the developer-defined security-critical code. As long as the code inside a compartment does not leak any secrets, Iso-X guarantees strong protection of the compartment's internal memory from any malicious OS or hypervisor activity. Specifically, a compartment becomes protected once it has been created, the compartment code and data have been added, and the compartment is attested and sealed. At the same time, the process of compartment creation by itself need not be secure and any entity is allowed to create compartments.

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. In addition, Iso-X is not inherently vulnerable to Iago attacks [9]. To protect against them, the code running in the compartment must treat all information from the OS as potentially malicious.

We do not consider denial-of-service (DoS) in our threat model, because it is already trivial for a malicious OS to deny service to an application or compartment. We also do not consider side channel attacks; several techniques exist to protect against them [16], [49].

We assume that the hardware TCB of Iso-X is limited only to the microprocessor, physical memory (DRAM), and system buses. In particular, 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 [18], [42], [47]. We note that it is straightforward to amend Iso-X to consider physical memory to be untrusted, by incorporating well-known techniques for memory integrity verification (such as Merkle trees) and encryption [8], [45].



(a) Physical memory layout

(b) Virtual memory layout

Figure 2: Iso-X Memory Layout

comp_base	**comp_size**	**page_count**	**comp_hash**	**cpt_base**	**cpt_size**
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

Table I: Format of a CT Entry

III. ISO-X DESIGN AND IMPLEMENTATION

This section describes the Iso-X architecture and implementation. We start with a design overview and follow with the data structures used to support compartments. Then, we describe the runtime interfaces exposed by Iso-X: the instructions used to implement compartment mode and the transition to and from the compartment.

A. *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. A high-level comparison of the traditional system organization and the Iso-X system is shown in Figure 1.

B. *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. Therefore, memory accesses to check the CMV bits in memory (or caches) are only required on a TLB miss.

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 I. 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 that is mapped to compartment space 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.

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 III-E 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 2.

C. *Iso-X Operations and Instructions for Compartment Management*

To support compartment management operations, several new instructions are added to the ISA and are directly sup-

Instruction	**Arguments**	**Priv Mode**	**Comp Mode**
① COMP_INIT	comp_id, comp_base, comp_size, cpt_base, cpt_size	Yes	No
② CPAGE_MAP	comp_id, virt_addr, phys_addr, page_permission_bits	Yes	No
③ COMP_ENTER	comp_id	No	No
④ COMP_ATTEST	None	No	Yes
⑤ CPAGE_REVOKE	comp_id, phys_addr	Yes	No
⑥ COMP_RESUME	comp_id	Yes	No

Table II: Iso-X Instructions

ported by the Iso-X hardware. These instructions, along with the mode in which they can be accessed, are summarized in Table II. Figure 3 depicts abstract algorithmic descriptions of these instructions and their impact on the processor state. The usage of these instructions is demonstrated in the following section.

D. 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 COMP_INIT ①.

To execute the COMP_INIT instruction, the Iso-X hardware zeroes out the CT entry indexed by *comp_id*. The entry is filled in based on the parameters of the COMP_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.

Notation Used in Figure 3	
$\mathbb{M}[x:y]$	Access to physical memory range from x to y
$\mathbb{H}(h,\ldots)$	Extend hash h with new data
$\mathbb{S}$	Sign with CPU private key
PAGE_SIZE	Platform specific memory page size
META_PAGE[]	Access to compartment metadata page
PC	CPU program counter

```
① COMP_INIT (comp_id, comp_base, comp_size,
cpt_base, cpt_size):
CT[comp_id].page_count,comp_hash←0
𝕄[cpt_base:cpt_base+cpt_size]←0
CT[comp_id].cpt_base,cpt_size←
  cpt_base,cpt_size
CT[comp_id].comp_base,comp_size←
  comp_base,comp_size

② CPAGE_MAP (comp_id, virt_addr, phys_addr,
page_permission_bits):
ASSERT(¬CMV[phys_addr])
CMV[phys_addr]←true
CT[comp_id].CPT[virt_addr]←phys_addr
CT[comp_id].comp_hash←ℍ(CT[comp_id].comp_hash,
  𝕄[phys_addr:phys_addr+PAGE_SIZE],
  virt_addr, page_permission_bits)
CT[comp_id].page_count+1

③ COMP_ENTER(comp_id):
CCR.ID←comp_id
CCR.CT←CT[comp_id]
CompMode←true
PC←CCR.CT.comp_base

④ COMP_ATTEST:
CCR.CT.comp_hash← ℍ(CCR.CT.comp_hash,
  META_PAGE[COMP_PUB_KEY])
META_PAGE[CERTIFICATE]← 𝕊(CCR.CT.comp_hash)

⑤ CPAGE_REVOKE(comp_id, phys_addr):
𝕄[phys_addr:phys_addr+PAGE_SIZE]←0
CMV[phys_addr]←false
CT[comp_id].page_count-1

⑥ COMP_RESUME(comp_id):
CCR.ID←comp_id
CCR.CT←CT[comp_id]
PrivilegedMode←false
CPU_REGISTERS←META_PAGE[CONTEXT]
CompMode←true

⑦ EVENT_COMP_LEAVE:
META_PAGE[CONTEXT]←CPU_REGISTERS
CPU_REGISTERS←0
CT←CCR.CT
CompMode←false
```

Figure 3: Algorithmic description of Iso-X instructions and events

Populating the created compartment with memory pages is accomplished using another Iso-X instruction called CPAGE_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 CPAGE_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, preventing attacks such as mapping legitimate data to incorrect virtual pages, or mapping pages with malicious permissions. Finally, the *page_count* field of CT is incremented.

To enter a compartment from an untrusted address space, the COMP_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 pre-defined location. The register state remains intact during this transition to allow the UP to pass data to the compartment.

Note that there is no need to authenticate that the request to enter the compartment comes from the process that created it. This is because the process itself is part of the untrusted partition, and thus there is no difference from the security standpoint whether the compartment was created by the process that owns it or by some malicious entity. If a malicious entity creates a malicious compartment, it can be detected by the compartment attestation mechanisms described in the next subsection.

E. Attesting Compartments and Building Trusted Channels

The code executed inside an Iso-X compartment does not trust any external data. Therefore, to communicate with other trusted entities that are located outside of the compartment (such as other compartments, or trusted resources outside the system), the compartment leverages a secure communication library which uses standard cryptography techniques to build a secure channel through the untrusted partition [31]. This functionality is similar to that used in HSM devices [40] to create trusted communication channels. Figure 4 illustrates the process of using a secure library – it shows how the Iso-X compartment transmits secret data to a trusted resource on the network.

After a compartment is created, the Iso-X system provides the opportunity to attest the compartment's integrity to outside entities. For this purpose, another Iso-X instruction, `COMP_ATTEST` ④, is used. The purpose of this instruction is to sign data that represents a compartment's identity with the processor key, providing evidence of the fact that the compartment contains only legitimate code and data. Iso-X uses a public/private key pair that is uniquely generated at CPU manufacturing time. This means any external entities that rely on compartments are assured that they are executing on genuine Iso-X hardware and not within an emulator. In addition, this allows the CPU to create a certificate for each compartment which can be used to verify the identity of a compartment and the CPU it is running on.

The secure communication model of compartmentalized applications relies on the secrecy of a compartment's public/private key pair. 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.

When the `COMP_ATTEST` instruction is executed from within the compartment, the Iso-X hardware combines the *comp_hash* field of CCR.CT with the compartment's public key from the metadata page and signs the resulting data with the CPU's private key. The resulting certificate is then placed on the compartment metadata page. It can then be used by the compartment to prove its identity to any outside entities. The correct certificate also proves the integrity of the compartment's initial state. Including the public key of the compartment in the signed data mitigates man-in-the-middle style attacks, as the OS does not know the generated private key, nor can it reproduce a valid compartment certificate.

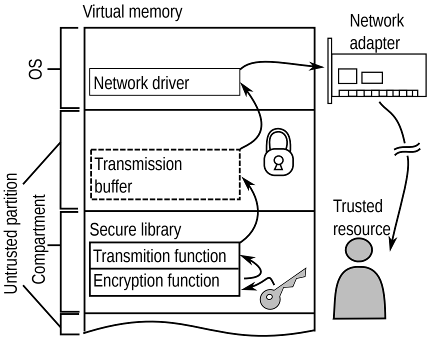


Figure 4: Compartment communicating securely to a remote resource

This model allows the outside entity, which is verifying the compartment, to also be guaranteed of the integrity of the compartment's public key. After `COMP_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.

F. Revoking Pages and Destroying Compartments

Revoking compartment pages and subsequently destroying compartments is accomplished via the `CPAGE_REVOKE` ⑤ instruction. The `CPAGE_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 free compartment pages.

G. 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 `EVENT_COMP_LEAVE` ⑦. 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 compartment mode. This mechanism guarantees that neither the OS, nor the untrusted partition can view or modify the registers of a compartment during interrupts, or when the compartment

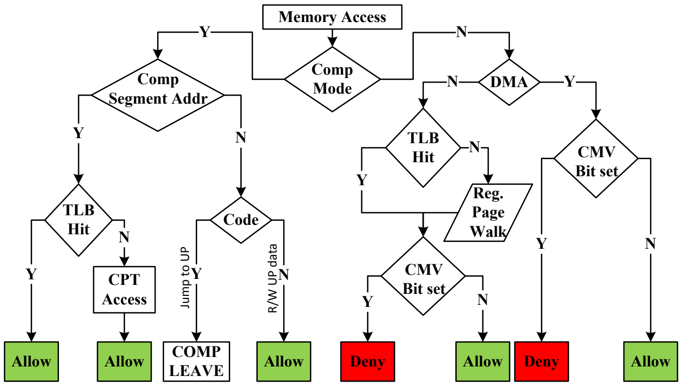


Figure 5: Memory Access Data-flow in Iso-X

phase finishes its execution.

To resume the compartment execution after it has been context switched, another Iso-X instruction – COMP_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.

H. Performing Memory Accesses in Iso-X

In this section, we describe how the Iso-X system performs memory accesses while executing in both compartment and regular modes. The memory data-flow chart illustrating this process is depicted in Figure 5.

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 EVENT_COMP_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.

Note that memory accesses in either mode involve the extra delay of accessing physical memory to read the CMV bits. We optimize most of this delay away by caching the CMV bit in the TLB; CMV bits are accessed in memory only on TLB misses.

For direct memory access using physical addresses, CMV checks are also performed to ensure that none of the pages used for DMA belong to a compartment: DMA accesses to a compartmentalized page are not allowed since they are controlled by the system software and could be abused.

I. Accessing OS services from the Compartment

System calls, network, and disk access require critical data to be operated on outside the compartment. Our solution relies on using the UP as an intermediate buffer between the TP and the destination device. To send a network packet or read/write to the disk, the TP writes the encrypted data to the UP and then uses a standard syscall to perform the I/O. This principle is demonstrated in Figure 4.

J. 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. The Iso-X hardware accomplishes this through the COMP_SWAP_PREP instruction, which is available to system.

COMP_SWAP_PREP *(comp_id, virt_addr)*. First, the TLB entry that covers the virtual page *virt_addr* is invalidated. Second, the page to be swapped out is measured and the hash value is saved in the now unneeded CPT entry. Either a 128-bit or a 256-bit hash can be used. The size of a CPT record has to be adjusted to allow storing measurements of the desired size. Next, the page is encrypted with a symmetric key. This key is randomly generated by the hardware on boot and stored in a register that is not software accessible. Then, the valid bit in the CPT is cleared as

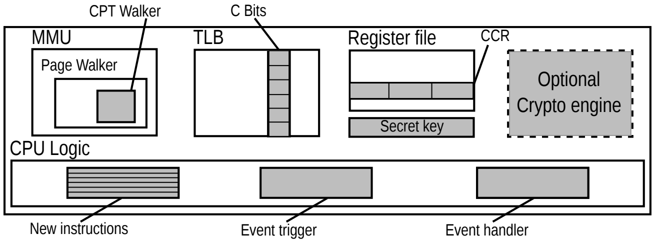


Figure 6: Processor Modifications Required by Iso-X

well as the corresponding CMV bit. 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.

Swapping the page back into memory also requires cooperation 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 regular ones. After a compartment page is moved to memory, the OS executes the `COMP_SWAP_RET` instruction to complete the swapping process as follows.

`COMP_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.

K. *Summary of Hardware Changes*

Figure 6 summarizes all of the required CPU hardware changes. These include the extra CCR register, extra *C* bits in the instruction and data TLBs, support for the new ISA instructions to handle compartments, the addition of a public/private key pair, and also logic to detect and handle the `EVENT_COMP_LEAVE` event. In addition, Iso-X performs hashing and signing operations for attestation and also encryption for optional swapping in hardware.

IV. Related Work

Now that we have described the design and implementation of Iso-X, we are in a position to compare it to previous efforts that proposed secure processor designs to protect application secrets from system compromise. Many solutions [29], [33], [37], [39], [52] rely on some trusted software module without hardware support. Unless this software is formally verified, it is difficult to guarantee the security of such schemes; the trusted software base can itself be compromised by the attackers. In addition, software approaches often incur substantial performance overhead. In the remainder of this section, we focus mostly on hardware-supported solutions with the exception of Inktag [21] and Virtual Ghost [12] as representatives of recent state-of-the-art software-only isolation schemes.

A. *Hardware-Assisted Isolation*

Hardware-assisted solutions are motivated by better security and performance [8], [11], [26], [27], [34], [44]. Table III summarizes the differences between Iso-X and other hardware-assisted solutions, as well as to Inktag [21] and Virtual Ghost [12] which are software only solutions. The *granularity of protection* refers to the minimum unit for isolation. *System software in TCB* indicates whether the solution includes software as part of the Trusted Computing Base. *Limited isolated execution* distinguishes between solutions that allow only limited operations in the isolated environment from those that do not. *Hardware Attestation mechanism* indicates whether the solution supports hardware attestation to verify the initial state of the compartment. *Dynamic protected space* refers to whether the solution provides the ability to flexibly allocate and grow the isolated regions. *Requires encrypted executable* refers to whether executables must be encrypted with the processor's hardware key. Finally, *secrets can reside anywhere* indicates whether any region of memory can be protected or if the system requires that protected data and code must be placed in a special region of memory. The related efforts are listed in the table in order of granularity of protection starting from the largest. In the remainder of this subsection, we examine some of these solutions in more detail.

The Secret Protection (SP) architecture [17], [26] 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 [8], 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 [11] 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. In the Iso-X system, neither the hypervisor, nor the guest operating systems are part of the trusted computing base. Inktag [21] is a recent software-only solution that uses *para-verification*, 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 system against the attacks launched by a malicious hypervisor in virtualized systems. These include HyperWall [47], H-SVM [42] and HyperCoffer [51]. The granularity of isolation for these solutions is the Virtual Machine.

	Granularity of protection	System software in TCB	Limited isolated execution	Hardware Attestation mechanism	Dynamic protected space	Requires encrypted executable	Secrets can reside anywhere
Iso-X	Virtual memory region[a]	N	N	Y	Y	N	Y
Academia:							
Hyperwall [47]	Virtual machine	N	Y	Y	Y	N	Y
H-SVM [42], Hypercoffer [51]	Virtual Machine	Y[b]	N	Y	Y	N	Y
Bastion [8], InkTag [21], SecureMe [11]	Process	Y	N	N	Y	N	Y
XOM [27]	Process	N	N	N	Y	Y	Y
AEGIS [46]	Process	N	N	Y	Y	N	N
Virtual Ghost [12]	Virtual memory region	Y/N[c]	N	N	Y/N[d]	N	N
OASIS [34]	Cache-as-RAM module	N	Y	Y	N	N	N
SP [26]	Trusted module (one per system)	N	N	N	N	N	Y
Industry:							
SecureBlue++ [7]	Process	N	N	N	Y	Y	Y
SGX [30]	Virtual memory region	N	N	Y	N	N	N

Table III: Comparing Iso-X with Related Efforts on Isolated Execution (Green is a strength, red is a limitation)

[a] Iso-X can isolate any arbitrary component of a process; a process can have multiple compartments.

[b] For running applications H-SVM and Hypercoffer require a trusted guest OS. In addition, Hypercoffer has a thin trusted software.

[c] The compiler remains trusted to instrument the OS kernel.

[d] The ghost memory segment itself is not dynamically expandable, but can be used dynamically.

The effort of [18] proposed an architecture based on non-inclusive memory permissions, thus not automaticaly 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 prevent a range of emerging attacks by placing hardware (rather than the hypervisor) in charge of security-critical decisions, such as setting the access rights to the code and data.

XOM [27] proposes execute-only memory that allows instructions to be executed, but not manipulated in other ways. XOM code is encapsulated in compartments to isolate different applications; each compartment is a process. Compartment's code is decrypted using the compartment session key. The code and data only leave the compartment in encrypted form. While the high-level idea of compartments in XOM is similar to Iso-X, the mechanisms used in Iso-X rely on hardware support for memory permissions rather than encryption.

AEGIS [44], [46] provides a secure execution environment where any physical or software tampering becomes evident. One of the AEGIS implementations relies on a security kernel, while another implementation does not [44]. Both designs require that the application remains in secure mode throughout its execution. Improvements to the design [46] rely on a trusted security kernel as part of the TCB to relax this requirement and allow the tamper-resistant state to be maintained while unprotected code executes.

OASIS [34] supports isolated execution on minimally-modified commodity CPUs by using physically-unclonable functions to create unique cryptographic keys for security-sensitive applications. It also uses the recently proposed Cache-as-RAM mode [28], [48], where the cache subsystem is re-purposed as a general-purpose memory area for isolated execution. As such, the size of the compartments is limited by the cache size, and performance may be affected by the reduced availability of the conventional cache.

Recently, designs for compartmentalized and isolated execution developed by industry have been introduced indicating significant commercial interest in this execution model. IBM introduced an isolated execution design called SecureBlue++ [7]. SecureBlue++ is designed specifically for the PowerPC architecture. This mechanism cannot combine secure and insecure code in a single process. Another related industry development is the recent introduction by Intel of its SMEP/SMAP mechanism to protect some user-level pages from being executed and/or accessed by supervisor mode code. This support is available in Ivy Bridge and later. However, to support proper functionality, SMAP has to be toggled on and off to allow the OS to access the user-space buffers. Trusting the OS to toggle SMAP removes any protection against a malicious OS, which is at the core of our threat model.

Finally, Intel's recent SGX security extension is perhaps the most significant recent development in hardware-supported security in industry [4], [20], [30]. SGX is scheduled to appear in Intel products in the future and it 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.

B. *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. The primary differences between Iso-X and SGX are in the following areas.

Compartment memory management and performance predictability: SGX requires all of the compartment'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 cache, which implies some limitations. For example, if the EPC cannot fit the memory pages for all compartments, the OS would need to evict compartment 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). Note that over-provisioning of the EPC would lead to wastage of DRAM space. 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.

Memory Access Latencies and Performance in Secure Mode: The second major difference is in the data-flow (and associated latencies) of memory operations. In SGX, every memory access in the secure mode requires checking the EPC Map (EPCM) structure located in reserved memory. While the published details of SGX provide no low-level details of this operation, it is reasonable to assume that EPCM bytes will also be cached in the CPU caches, to avoid DRAM accesses. However, the size of the EPCM can put pressure on the caches; based on our understanding of SGX, each page requires an entry of size 79 bits [30]. In addition, EPCM accesses are required in SGX after every TLB access (hit or miss) in secure mode. Although it may be possible to amortize these checks, the security implications are not clear. In contrast, Iso-X only accesses CMV bits on TLB misses because Iso-X verifies the mappings rather than the page permissions. In summary, it is possible that SGX will experience higher performance overhead while executing in secure mode due to these effects. On the positive side, SGX requires no additional checks in non-secure mode, so its performance in non-secure mode could be slightly better than Iso-X.

Dynamic Extensibility of Compartment Memory: In SGX, once the compartment is created, it needs to be sealed. After that, new pages cannot be added to the compartment. This restricts the programming model, leaving application designers to choose between over-allocating their compartment size (and hence creating internal fragmentation) or risking running out of space if they need it. In contrast, Iso-X allows the dynamic addition of new, empty pages to the compartment.

V. Performance Evaluation

With respect to performance, we evaluate two sources of overhead: (1) overhead due to the extra memory permission checks that occur with every memory access; and (2) One-time or infrequent overhead associated with events such as compartment creation, destruction, and page swapping.

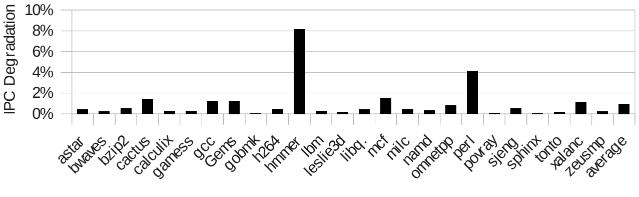


Figure 7: Iso-X Performance Degradation

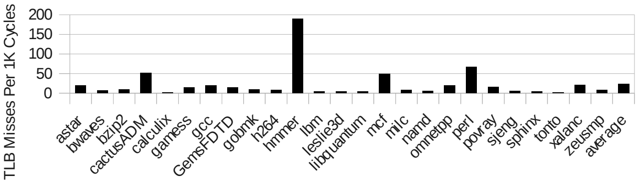


Figure 8: Combined TLB Miss Rates

A. 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 [43] using the MARSSx86 full system x86-64 simulator [2]. The simulated processor configuration is depicted in Table IV. 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 also stored in the processor caches, we modeled this impact in our evaluations. We expect the performance loss and the extra cycles encountered during execution 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 III), 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 7 shows the decrease in the commit IPCs of the 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 0.97% on the average in both secure and non-secure mode. The largest loss among individual benchmarks is that of hmmer at about 8%. Figure 8 depicts the combined miss rates for both data and instruction TLBs for the Iso-X system. As the additional delays in Iso-X are encountered during TLB misses, there is a strong correlation between

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 IV: Configuration of the Simulated x86-64 Processor

Operation	Actions(s)	Cost (Cycles)
Create	System Call	138
	Find Free Comp.	2,107
	`COMP_INIT`	2
	Total	2,247
Populate	Hash Page(s)	6.172M
Attest	Sign Hash	1.241M
Revoke Page	Zero Page	596
Destroy	Revoke All Pages	52,379
Interrupt	`EVENT_COMP_LEAVE`	52
Resume	`COMP_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 V: Compartment Operation Overheads

TLB miss rate and IPC loss.

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 [25] 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.

B. 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 V. This table shows the number of cycles that each compartment operation takes. In addition, each operation is broken down into the actions that it requires and the costs of these individual actions are also reported. These figures 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 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 [36] 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 V the most expensive ones by far 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, these overheads can be tolerated since they are on the order of at most a few milliseconds for each compartment. All other overheads presented in Table V are much smaller.

Some of these operations can be significantly 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 [32], which is about 10x faster than the software implementation reported in Table V. As another example, AES encryption performed on a separate engine clocked at 340MHz requires 0.69 cycles/byte [19], 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.

C. 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

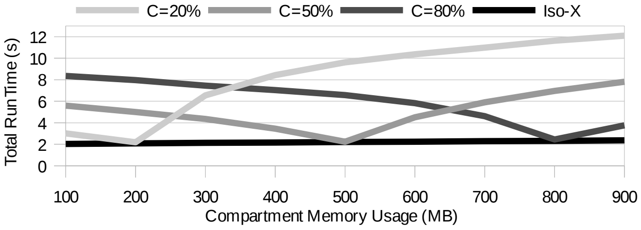


Figure 9: Effect of Static Memory Provisioning

be difficult. For example, 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 9) 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. The scenario represents the best case advantage for Iso-X; under many operating conditions the memory pressure may be lower, leading to a margin for error in provisioning the two memory regions. However, if the system uses magnetic drives instead of SSD, the impact of incorrect provisioning will be substantially higher.

VI. Iso-X Applications Examples

Some classes of applications can directly benefit from the Iso-X system by storing their secrets and performing computations on sensitive data in isolated compartments. The examples of such applications include banking and e-commerce applications, digital rights management applications, password managers, and disk encryption software. It is also interesting to explore using hardware compartments to provide stronger guarantees to support software isolation mechanisms (e.g., Java Isolation [6]). In addition to these, we describe two more scenarios where Iso-X can provide benefits.

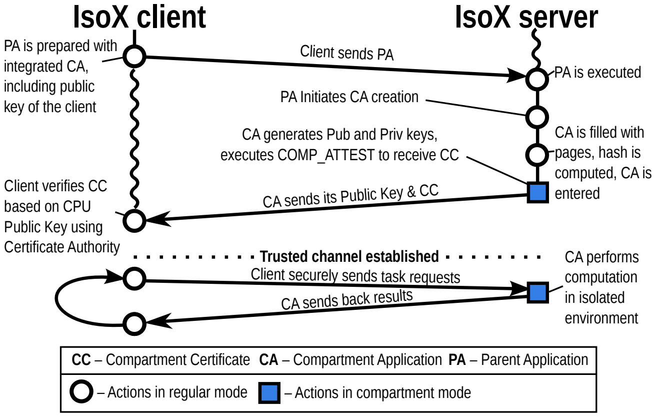

IsoX client
IsoX server
PA is prepared with integrated CA, including public key of the client
Client sends PA
PA is executed
PA Initiates CA creation
CA is filled with pages, hash is computed, CA is entered
CA generates Pub and Priv keys, executes COMP_ATTEST to receive CC
Client verifies CC based on CPU Public Key using Certificate Authority
CA sends its Public Key & CC
Trusted channel established
Client securely sends task requests
CA sends back results
CA performs computation in isolated environment
CC – Compartment Certificate
CA – Compartment Application
PA – Parent Application
○ – Actions in regular mode
■ – Actions in compartment mode

Figure 10: Usage Flow in Cloud Scenario

A. *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. Figure 10 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 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)[40]. 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.

B. Secure Machine Attestation:

Compartmentalized code 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.

VII. Limitations of Iso-X

While Iso-X provides a secure execution environment, there are restrictions from the standpoints of both applications and system design.

Applications taking advantage of Iso-X should be partitioned in such a way that no data flow exists from a trusted to an untrusted partition. In addition, compartment code should not be directly controlled by any external untrusted entity, such as the UP or the operating system. For this reason, the current design of Iso-X is more applicable to the cloud scenario, as exemplified by Figure 10. To adapt Iso-X to desktops and mobile devices, it needs to be augmented with mechanisms for trusted I/O. While trusted I/O is out of the scope of this paper, it was recently addressed by other works [53]. Note that if only a partial isolation of compartments is required, then some of these conditions can be relaxed.

In terms of system design, the current Iso-X implementation relies on the presence of a reserved memory region which is only accessible to the Iso-X hardware. In addition, the design disallows DMA accesses to compartment pages, as defined by the compartment membership vector (CMV).

VIII. 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 being protected by 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.

The current design represents a first effort at developing the Iso-X architecture. Our future work plans include support for multithreaded operation in compartment mode as well as developing applications to use the architecture.

Acknowledgment

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] 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.

[5] Anonymous. Xbox 360 Hypervisor Privilege Escalation Vulnerability, 2007. Available online: http://www.securityfocus.com/archive/1/461489.

[6] 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.

[7] R. Boivie and P. Williams. Secureblue++: Cpu support for secure executables. 2013.

[8] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *Proceedings of HPCA*, 2010.

[9] S. Checkoway and H. Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264. ACM, 2013.

[10] 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.

[11] 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)*, June 2011.

[12] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proc. ASPLOS*, 2014.

[13] CVE-2007-4993: Xen guest root can escape to domain 0 through pygrub, 2007.

[14] CVE-2007-5497: Vulnerability in XenServer could result in privilege escalation and arbitrary code execution, 2007. Available online:http://support.citrix.com/article/CTX118766.

[15] CVE-2008-2100: VMware Buffer Overflows in VIX API Let Local Users Execute Arbitrary Code in Host OS, 2008.

[16] 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*, June 2012.

[17] J. Dwoskin and R. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of CCS*, 2007.

[18] 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.

[19] 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*, pages 60–63. ACM, 2005.

[20] 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.

[21] O. Hofmann, S. Kim, A. Dunn, M. Lee, and E. Witchel. Inktag: Secure applications on an untrusted operating system. In *Proceedings of ASPLOS*, 2013.

[22] V. Kemerlis, G. Portokalidis, and A. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 39–39. USENIX Association, 2012.

[23] C. King and C. Beal. Csi kernel: Finding a needle in a multiterabyte haystack. *Software, IEEE*, 29(6):9–12, 2012.

[24] K. Kortchinsky. Hacking 3D (and Breaking out of VMWare). In *BlackHat USA*, 2009.

[25] 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.

[26] 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*, pages 2–13. IEEE, 2005.

[27] 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.

[28] Y. Lu, T. Lo, G. Watson, and R. Minnich. Using cache as ram in linux bios, 2012. http://rere.gmgm.pl/mirq.

[29] J. Mccune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proc. ACM Eurosys*, 2008.

[30] 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.

[31] R. Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, Apr. 1978.

[32] 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)*, 5(1):2, 2012.

[33] 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.

[34] 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.

[35] 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.

[36] polarssl, 2014. Accessed May 2014 at https://polarssl.org/.

[37] 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)*, pages 1–20, 2008.

[38] J. Rutkowska. Introducing the Blue Pill, 2006. Available Online: http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html.

[39] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proc. of the 13th Usenix Security Symposium*, Aug. 2004.

[40] 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*, pages 120–123. IEEE, 2002.

[41] Cve details: The ultimate security vulnerability datasource, 2013. Accessed Nov. 2013 at http://cvedetails.com.

[42] S.Jin, J.Ahn, S.Cha, and J.Huh. Architectural support for secure virtualization under a vulnerable hypervisor. In *Proceedings of MICRO*, 2011.

[43] C. D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, 2007.

[44] 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.

[45] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of MICRO*, 2003.

[46] 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.

[47] J. Szefer and R. Lee. Architectural support for hypervisor-secure virtualization. In *Proceedings of ASPLOS*, 2012.

[48] A. Vasudevan, , J. McCune, J. Newsome, A. Perrig, and L. van Doorn. Carma: A hardware tamper-resistant isolated execution environment on commodity x86 platforms. In *Proc. ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[49] Z. Wang and R. Lee. A novel cache architecture with enhanced performance and security. In *Proc. International Symposium on Microarchitecture (MICRO)*, Dec. 2008.

[50] R. Wojtczuk. Subverting the Xen hypervisor. In *BlackHat USA*, 2008.

[51] 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.

[52] 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.

[53] Z. Zhou, V. Gligor, J. Newsome, and J. McCune. Building verifiable trusted path on commodity x86 computers. In *IEEE Symposium on Security and Privacy*, 2012.

[54] 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.