Enforcing Executing-Implies-Verified with the Integrity-Aware Processor[⋆]

Michael LeMay and Carl A. Gunter

University of Illinois, Urbana IL 61801, USA

Abstract. Malware often injects and executes new code to infect hypervisors, OSs and applications. Such malware infections can be prevented by checking all code against a whitelist before permitting it to execute. The *eXecuting Implies Verified Enforcer (XIVE)* is a distributed system in which a kernel on each target system consults a server called the *approver* to verify code on-demand. We propose a new hardware mechanism to isolate the XIVE kernel from the target host. The *Integrity-Aware Processor (IAP)* that embodies this mechanism is based on a SPARC soft-core for an FPGA and provides high performance, high compatibility with target systems and flexible invocation options to ensure visibility into the target system. This facilitates the development of a very small trusted computing base.

1 Introduction

Hypervisors, OSs, and applications continue to be infected by malware [11]. One common result of compromise is the execution of foreign code on the target system. Foreign code can be injected directly into a process as a result of a memory corruption bug, or it can be a separate program that is downloaded and installed on the machine as a part of the attack. Eliminating this foreign code would severely limit a successful attack.

One way to prevent foreign code from running on a system is to whitelist the code in legitimate applications and refuse to run any code that is not on this whitelist, thus enforcing the *eXecuting* $\rightarrow$ *Verified* property on all code. This conceptually straightforward approach has been difficult to implement in practice. It is challenging both to identify legitimate applications and to enforce the resultant whitelist. We assume that legitimate applications are known in advance and focus on enforcement in this paper. Past efforts exhibit deficiencies in some combination of the following requirements: *1) Isolation*, making the integrity enforcer vulnerable to compromise; *2) Visibility*, reducing their capability to detect compromises in the target system; *3) Performance*, making them impractical for some applications; *4) Compatibility*, necessitating that the target be substantially modified.

[⋆] In: Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST 2011), Pittsburgh, PA, USA.

Patagonix manipulates *Memory Management Unit (MMU)* page tables to cause a trap to the Xen hypervisor whenever an unverified page in a *Virtual Machine (VM)* is about to be executed [13]. Xen provides many features, but has a correspondingly large *Trusted Computing Base (TCB)* ($\sim$ 230K lines of code [4]) that may be unable to enforce isolation. Even target systems that can use minimalistic, security-oriented hypervisors, like SecVisor [18], may suffer from virtualization-related performance degradation. Furthermore, some virtualization approaches require substantial changes in the target system's code to make it compatible with the hypervisor.

Intel *System Management Mode (SMM)* can be used to overcome some of these limitations, since it provides hardware separate from the MMU to set up an isolated execution environment [4, 20]. However, the confidentiality and integrity of SMM handlers can be tricky to guarantee due to the complex interactions between the various system components involved in implementing SMM [9]. Furthermore, some system state is invisible in SMM, and SMM can only be triggered by an electrical signal at one of the processor's pins or by writes to a control register. Code has been observed to execute about two orders of magnitude more slowly in SMM compared to protected mode [4].

We propose a new hardware mechanism that addresses the limits of other approaches. The *Integrity-Aware Processor (IAP)* is an extended SPARC processor that provides an isolated execution environment for an *integrity kernel* that can enforce eXecuting $\rightarrow$ Verified or provide other functionality. Although the integrity kernel shares many processor resources with the target, IAP stores the integrity kernel in a completely separate address space from the target and does not permit the target to initiate any data transfers between the two spaces. On the other hand, the integrity kernel has full visibility into the target. IAP stores the entire integrity kernel on-chip to minimize access latency. Thus, integrity kernel code runs at least as fast as code in the target system, and it can be invoked with the same overhead as a native trap handler. IAP also incorporates hardware accelerators for essential cryptographic primitives.

IAP transfers control to the integrity kernel in response to several configurable conditions, including attempts by the processor to execute code that has not been verified by the integrity kernel. It includes hardware structures to track code that has been verified and to detect attempts to modify it after it has been initially verified, subsequently causing it to be re-verified before it is re-executed. IAP monitors individual pages of memory in the virtual address space of each process, or the physical address space when the MMU is disabled. These features permit the integrity kernel to enforce eXecuting $\rightarrow$ Verified without relying on the configuration of the MMU page tables, further simplifying the TCB.

We developed the *eXecuting $\rightarrow$ Verified Enforcer (XIVE)* to demonstrate the extent to which IAP can reduce the TCB of the integrity kernel. The XIVE whitelist is located on a centralized *approver* connected to the network, to minimize the complexity of the XIVE kernel and to simplify whitelist updates. We implemented XIVE for an FPGA-based IAP end node, the approver on a commodity Linux host, and connected the two using 100Mbps Ethernet. The XIVE kernel comprises 859 instructions, and successfully protects a Linux target system that has been slightly modified so that it is efficient to monitor and so that it tolerates sharing its physical network interface with the XIVE kernel.

The rest of this paper is organized as follows. §2 contains the rationale for our design decisions. §3 discusses implications and limitations of the design, including potential future directions. §4 evaluates our implementation. §5 explains the relationship of this paper to related work. §6 concludes the paper.

2 Design

2.1 Threat Model

We adopt the Dolev-Yao model for attacks on the LAN hosting XIVE [8]. The attacker is permitted to use *Direct Memory Access (DMA)* to modify memory. We disallow physical attacks.

2.2 Hardware

IAP is based on the LEON3 SPARCv8 soft core by Gaisler Research. We based our design on an instantiation of the LEON3 that implements a 7-stage pipeline, separate data and instruction caches *(D-cache* and *I-cache*, respectively*)*, an MMU with a split *Translation Lookaside Buffer (TLB) (D-TLB* and *I-TLB)* and a hardware page table walker, and an AMBA 2.0 AHB system bus. SPARC processors support several distinct address spaces. Some of the address spaces refer to main memory and peripherals, and others are used to configure the processor or access special features. Our changes to the LEON3 are mostly concentrated in the pipeline, cache, and MMU subsystems. IAP partially relies on the SPARC coprocessor interface to interact with the pipeline.

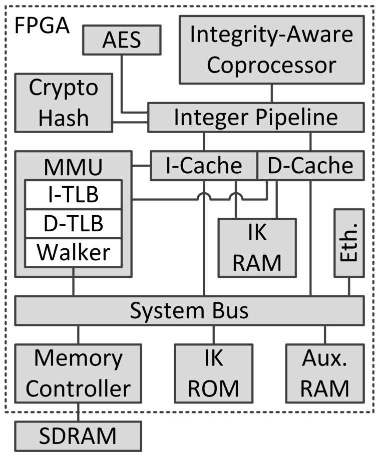


Fig. 1: *Internal connectivity of IAP components.*

Figure 1 illustrates the internal connectivity of the major components in IAP, each of which will be discussed below. The central addition to the processor is a region of on-chip RAM (called the *integrity kernel RAM*) that is only accessible in integrity kernel mode, which is analogous to supervisor mode and possesses strictly greater access privileges. Integrity kernel RAM can be accessed by the I-cache and the D-cache, through a dedicated port for each. Integrity kernel RAM occupies a dedicated address space. Accesses to integrity kernel RAM are not mediated by the MMU since the entire integrity kernel is trusted. IAP contains a ROM from which the integrity kernel is loaded into integrity kernel RAM immediately after the processor is reset. Control is then transferred to the first instruction in the integrity kernel.

Attempts to execute unverified instructions are detected within the I-TLB and the I-cache. Each I-TLB entry contains a *V bit* (for "Verified") that is cleared whenever the entry is inserted into the I-TLB or the memory region that it encompasses is written. However, TLBs are only consulted when the MMU is enabled, so IAP also includes a separate set of *non-MMU V bits* that each map to a portion of the physical address space. V bits allow an integrity kernel to ensure that specific regions of virtual memory (in this context the physical address space constitutes a separate virtual space) have been verified before any of their contents are executed as instructions. The specific type of verification to be performed must be implemented by each integrity kernel. IAP provides facilities so that the integrity kernel can selectively and efficiently set or clear any number of V bits associated with a specific physical or virtual address region in constant time.

XIVE minimally requires hardware support for detecting and handling two types of events. Additional types of events could be supported by future versions of IAP to support other types of integrity kernels. The process for handling each event in hardware is described in Listing 1. Certain aspects of the event handling warrant further explanation, which we now provide. Native SPARC traps automatically allocate an empty register window for the handler by adjusting the register window pointer. However, integrity kernel traps may be invoked within a native trap, so they do not adjust the SPARC register window pointer. Thus, storing the current and next program counter values in local registers as usual would overwrite program data. In this case, IAP stores those values in shadow registers instead. Their contents shadow the corresponding normal registers for "long jump" and "return from trap" instructions. By only shadowing the registers for those instructions, we ensure that the registers can otherwise be used normally by the integrity kernel. Neither of the new *Processor State Register (PSR)* bits defined in Listing 1 is visible outside the integrity kernel.

The circuitry that fetches instructions for the pipeline is complex, and it is not immediately obvious that IAP correctly requires all instructions to be verified before being executed by the pipeline. We do not have space in this paper to provide a complete argument that it does so, but we note that it uses the D-cache bus snooping circuitry to detect write accesses by all bus masters to memory and appropriately update the affected V bits. It also contains circuitry to handle the corner cases that arise when bus masters write to memory that is currently being accessed by the I-cache in its various modes of operation.

IAP implements two cryptographic algorithms in hardware, since they are heavily used by XIVE and relatively expensive to implement in software. First, we selected the BLAKE hash routine for its open source VHDL implementation, status as a SHA-3 finalist, and good performance [19]. The implementation is sufficiently fast that hashing does not stall the pipeline, except during finalization. Second, IAP supports 128-bit AES. The AES implementation can cause pipeline stalls. However, it mostly operates in parallel with the pipeline, so stalls

Listing 1 *Hardware handling of individual events.*

```
procedure HandleEvent
    $\epsilon \leftarrow$ DetectEvent
    $\alpha \leftarrow$ PrescribeResponse($\epsilon$)
    if $\alpha = \langle Trap, \tau, \delta \rangle$ then
        Trap($\tau, \delta$)
    end if
end procedure
function DetectEvent
    if attempting to execute instruction from page with unset V bit then
        return $\langle HitVBit, \psi \rangle$                ▷ $\psi$ is the page information.
    else if $PC \in Breakpoints$ then      ▷ Integrity kernel can modify breakpoints.
        return $\langle HitBreakpoint, None \rangle$
    else
        return $None$
    end if
end function
function PrescribeResponse($\epsilon$)          ▷ Determine how to respond to the event.
    if $\epsilon = \langle \tau, \delta \rangle$ then
        return $\langle Trap, \tau, \delta \rangle$  ▷ Other types of responses could be supported in future
versions of IAP.
    else
        return $None$
    end if
end function
procedure Trap($\tau$, $\delta$)                        ▷ Trap to the integrity kernel.
    $ShadowPC \leftarrow PC$
    $ShadowNextPC \leftarrow NextPC$
    $PSR.IntegrityKernelMode \leftarrow True$  ▷ Controls access to processor resources and
causes the I-cache to fetch trap handler code from integrity kernel RAM.
    $PSR.TrapsPreviouslyEnabled \leftarrow PSR.TrapsEnabled$                ▷ Used to restore
$PSR.TrapsEnabled$ when exiting integrity kernel mode.
    (Continue invoking trap handler similarly to native trap handler.)
end procedure
```

can be avoided by inserting sufficient instructions between the point at which each AES operation is initiated and the point at which its output is used.

The Ethernet interface in IAP has been modified to support dual MAC addresses, which permits the integrity kernel to receive packets without forcing the interface into promiscuous mode or switching between MAC addresses. It places packets for both MAC addresses into a single DMA buffer.

2.3 Networking

There are three requirements that the network communications protocol between the integrity kernel and the approver must satisfy: *1) Security*, to prevent eavesdropping and to detect attempts to modify packets. *2) Low latency*, to minimize the amount of time that the end node is blocked waiting for a response from the approver. *3) Simplicity*, since it must be implemented in the integrity kernel, which has a very small codebase.

We constrain the approver to occupy the same local area network as the end node to minimize latency. This permits us to define a link layer protocol, called the *XIVE Network Protocol (XNP)*. Each XNP packet is 96 bytes long (excluding the 14 byte Ethernet header and four byte Ethernet CRC), and can be represented formally as a tuple $\langle \nu, \tau, \varsigma, \phi, \mu \rangle$ (excluding the Ethernet header). To prevent replay attacks, both approver and kernel nonces ν are drawn from a single, strictly increasing sequence. The packet type $\tau \in \{\textit{boot}, \textit{verify}, \textit{exit}\} \times \{\textit{request}, \textit{response}\}$. The sequence number ς is used to match responses to requests. The composition of the payload ϕ is specific to each type of exchange described below. EAX mode is used to encrypt and authenticate $\langle \nu, \tau, \varsigma, \phi \rangle$ with AES, and the resultant MAC value μ is then appended [5]. The XIVE kernel implements logic to resend packets after some timeout until a response is received. We now discuss specific XNP exchanges between the XIVE kernel $\mathcal{K}$ and the approver $\mathcal{A}$. Only the payload ϕ is depicted (when non-empty), but each transfer actually involves a complete packet.

The approver resets its internal representation of an end node's state, which we discuss later, whenever it receives a boot request from that node.

The XIVE kernel issues a page verification request whenever it detects an attempt to execute instructions from a page with an unset V bit: $\mathcal{K} \xrightarrow{\langle \tau_p, \gamma, \beta, \theta \rangle} \mathcal{A}$, where the page type τ_p is derived from the size of the page, the state of the MMU (enabled/disabled), and the processor mode (supervisor/user), γ is a unique identifier for the currently-executing process, β is the virtual base address of the page, and θ is the result of hashing the entire page using BLAKE. The XIVE kernel blocks the execution of the processor until it receives a response from the approver: $\mathcal{K} \xleftarrow{\langle \alpha \rangle} \mathcal{A}$, where α specifies what action the XIVE kernel must take next, which can be either to terminate the process or to resume its execution.

The XIVE kernel issues an exit request when it detects that a process context is about to be destroyed: $\mathcal{K} \xrightarrow{\langle \gamma \rangle} \mathcal{A}$, where γ is the same process identifier used in the page verification requests. This permits the process identifier to subsequently be reused. Note that this does introduce a certain level of trust in the target OS to achieve the full assurances possible with XIVE, since an OS that causes XIVE to issue an exit request without actually destroying the appropriate context can then potentially be permitted by XIVE to execute programs that contain unapproved combinations of pages. However, all code that is executed must still be recognized and approved.

Approver. The approver's roles include generating and maintaining a whitelist, a database of pre-shared AES keys, and a representation of the internal state of each active end node on the network. Our initial prototype is simplified in that it only communicates with a single end node.

To maintain a partial representation of the internal state of each end node, the approver creates an empty state when it receives an XNP boot request packet and updates that state when each new page verification request or process exit request packet is received. The current state of each node n can be represented as a function $\sigma : \mathit{\Gamma} \rightarrow 2^{\mathit{\Pi}}$ generated in response to a set of "current" page request packets ρ received from n, where $\mathit{\Gamma}$ is the set of all context numbers that are contained in ρ and $\mathit{\Pi}$ is the set of all approved programs:

$$\sigma(\gamma) = \bigcap_{\langle \tau_p, \gamma, \beta, \theta \rangle \in \rho} \left\{ \pi \in \mathit{\Pi} \;\middle|\; \langle \tau_p, \beta, \theta \rangle \in \pi \right\}$$

where $\langle \tau_p, \beta, \theta \rangle \in \pi$ iff the page with the specified characteristics is contained within program π. When $\sigma(\gamma) = P$, it means that the process identified by γ has only previously executed code that is contained in all programs in P. P may contain many programs, since different programs can contain pages of code with identical characteristics. A page request packet $\langle \tau_p, \gamma, \beta, \theta \rangle$ loses currency and is removed from ρ when the approver receives a process exit request packet $\langle \gamma \rangle$. We say that n has entered an unapproved state as soon as $\exists \gamma.\ \sigma(\gamma) = \emptyset$, meaning that the process identified by γ is not a recognized program.

To generate a whitelist, the approver can be operated in learning mode, in which it approves all pages of code and outputs a database representing all programs that were executed by n.

The approver software is implemented as a multi-threaded C++ program. One thread performs administrative functions, another receives, decrypts, and verifies packets, a third processes the received packets with respect to the system state database, and the final thread encrypts, authenticates, and transmits newly-generated packets.

2.4 XIVE Kernel

The kernel was coded entirely in assembly language and is described in Listing 2. It contains 859 instructions and uses 488 bytes of integrity kernel RAM and 2952 bytes of auxiliary RAM. Each trap handler saves all of the local registers, which on the SPARC are eight registers in the current register window, and the processor state register to integrity kernel RAM upon entry, which permits the handlers to freely use the local registers as well as instructions modifying the condition codes in the processor state register. Some trap handlers require more than eight registers. Thus, blocks of integrity kernel RAM are reserved to implement a pseudo-stack for register swapping, although this is not described here. The prototype kernel does not implement an actual stack, because it is simpler to directly address the reserved memory in the few instances that it is required.

Physical pages of the target's kernel code and shared libraries are mapped into multiple virtual address spaces for different processes, so XIVE by default hashes and verifies them in each address space. To reduce this unnecessary overhead, we implemented an optional hash caching mechanism that stores the hash for each I-TLB entry at the time that it is calculated, and re-uses that hash during subsequent attempts to verify the same physical page, as long as it has

Listing 2 *XIVE kernel*

```
procedure Boot    ▷ Obtains control immediately after every processor reset.
    $\eta_{dc} \leftarrow$ AddressOf(destroy_context)
    InitBP($\eta_{dc}$)    ▷ Initialize breakpoint to detect process context destruction.
    InitAES($\kappa$)    ▷ Initialize AES using key shared with the approver.
    XNPBoot    ▷ Perform XNP boot exchange.
    $\eta_{to} \leftarrow$ AddressOf(target OS)
    Jump($\eta_{to}$)    ▷ Transfer control to target OS.
end procedure
procedure HandleBreakpoint
    SaveLocalRegisters
    XNPProcessExit($\gamma$)    ▷ Perform XNP process exit exchange.
    RestoreLocalRegisters
end procedure
procedure HandleUnsetVBit($\psi$)    ▷ $\psi$ is the page information.
    SaveLocalRegisters
    SetVBit($\psi$, True)    ▷ Doing this first ensures that any DMA
accesses that occur during the subsequent verification operations are detected when
the processor resumes normal execution.
    $\theta \leftarrow$ BlakeHash($\psi$)    ▷ Hash entire page.
    $\alpha \leftarrow$ XNPVerifyPage($\psi, \theta$)    ▷ Perform XNP page verification exchange.
    if $\alpha = resume$ then
        RestoreLocalRegisters
        (Resume process execution.)
    else
        HaltProcessor    ▷ Our prototype
simply halts the target when it enters an unapproved state, rather than attempting
to selectively terminate the unapproved program.
    end if
end procedure
```

not been modified. Our prototype uses 2304 bytes of auxiliary RAM for the hash cache, although this is in fact vulnerable to manipulation by the target OS. A deployable implementation would place the hash cache in integrity kernel RAM.

The integrity kernel shares the Ethernet incoming DMA buffers with the target OS whenever the target OS has enabled the Ethernet interface. This makes it possible for packets intended for the target that arrive while the integrity kernel is active to eventually be received by the target. Otherwise, the integrity kernel uses a total of 512 bytes of auxiliary RAM for incoming DMA buffers. The integrity kernel always uses 136 bytes of auxiliary RAM as outgoing DMA buffers.

3 Discussion

Deployment and Management. In any XIVE-protected environment, the following elements must be deployed: *1)* IAPs to operate all programmable portions of the end nodes, *2)* At least one co-located approver server that is statically configured to be resistant to attacks, since it is unable to rely on XIVE protection itself, *3)* An integrity kernel ROM image for each IAP, and *4)* Pre-shared keys to permit authenticated, encrypted communication between each end node and the approvers. A variety of protocols can be devised to install keys and ROM images in end nodes, and may resemble the protocols that have previously been developed to securely configure sensor nodes, such as SCUBA [17].

Limitations. XIVE currently exhibits several limitations, which we now discuss in conjunction with likely remediation strategies. Since XIVE relies on network communications to approve the forward progress of each end node, attackers can deny service to legitimate users of those nodes by interfering with XNP. However, XNP only operates on the LAN, which reduces the ability of attackers outside of the LAN to launch denial-of-service attacks.

Control flow attacks can succeed without injecting new code into the target system [7]. Thus, XIVE does not prevent them. However, some types of control flow attacks, such as return-oriented-programming, can be prevented using address space layout randomization [6]. Each XIVE whitelist is specific to a particular address space layout. However, XIVE could potentially be adapted to work with randomized address spaces by causing the approver to issue a seed to control the randomization process on an end node. That seed could then be used by the approver to translate page verification requests.

Bytecode is never directly executed, but is instead processed by an interpreter or a *Just-In-Time (JIT)* compiler, the output of which is ultimately executed. Currently, XIVE would simply verify instructions executed within an interpreter, a JIT, and the output from the JIT. Certainly, it is desirable to verify interpreters and JITs, but it is likely to be infeasible to whitelist JIT outputs, since the JIT may dynamically generate various instruction streams. This can be handled by monitoring data reads and writes by recognized JITs, so that bytecode inputs can be verified and the output instruction streams intended to be executed in the future can be excluded from verification. Patagonix includes some elements of this approach [13].

One potential strategy for adapting XIVE to a multicore environment is to replicate most of its functionality on each core, designate one instance as the leader, and create local communication channels that connect all instances. Then, whenever an instance needed to communicate with the approver, it could route the communication through the leader, which would be the sole instance with access to the Ethernet interface.

Alternate Usage Models. It would be a simple matter to adapt the approver to approve rather than deny by default, and thus enforce a blacklist of known malware, permitting all other software to execute.

Alternately, by simply approving all software that is measured by the end node on the approver rather than preventing the execution of non-whitelisted software, the approver could then field remote attestation requests on the behalf of the end node. Some advantages of this approach over conventional remote attestation, such as that implemented by the *Linux Integrity Measurement Architecture (Linux-IMA)* [16], are that audit logs are maintained in a central location further reducing the TCB on end nodes, cumulative attestation can easily be provided [12], it conclusively reveals the presence of malware on infected target systems since malware is unable to block attestation requests, and the use of *Public-Key Cryptography (PKC)* is centralized so that fewer nodes must be upgraded if it is eventually broken.

4 Evaluation

Implementation. We synthesized IAP to run on the Digilent XUPv5 development board. We also ported general performance enhancements in IAP (selective I-TLB and I-cache flushing) to a reference version of the LEON3, which we used as the basis for a series of benchmarks. Both versions of the processor configure their I-cache with four sets of 32KiB each, a line size of eight words, and an I-TLB with 64 entries. Each of the TLB and cache structures in both processors implements a *Least Recently Used (LRU)* replacement policy. We synthesized both processors at a clock frequency of 50MHz using the *Xilinx Synthesis Tool (XST)* v.13.1. The reference utilizes 52% of the FPGA slices and 33% of the BlockRAMs and FIFOs. IAP utilizes 71% of the slices and 40% of the BlockRAMs and FIFOs. The prototype includes various debugging features that weaken its security, but these would be removed from a deployable implementation.

Linux 2.6.36 serves as the target OS in our prototype, hosting a Buildroot userspace environment. It was necessary to modify the Linux kernel to allocate smaller pages of memory containing kernel code and data (256KiB versus 16MiB), to reduce the size of the whitelist and the number of re-verification operations. We optimized the context switching mechanism to cause the processor to switch back to a dedicated kernel context upon entering the kernel. The I-TLB and I-cache include the context number in their entries' tags, so this reduces pressure on those structures.

Userspace programs and libraries also presented challenges for XIVE, because they mixed code and data pages, and the dynamic linkage process incrementally modified an executable region in each program. These issues caused the overhead from XIVE to increase to the point of infeasibility. We modified the linker scripts to re-align the program sections and thus avoid the first issue. We configured all programs at link-time to preemptively complete the dynamic linkage process when they are first launched to resolve the second issue.

For the purposes of our experiments, the Linux kernel and the userspace root filesystem are merged into a monolithic memory image that is loaded before the system boots. To permit the construction of a whitelist, we cause the image to fill a whole 16MiB page, as initially used by the kernel, and zero-fill all unused space within the image.

TCB Size. XIVE was constructed to demonstrate that a very small integrity kernel is capable of enforcing eXecuting $\rightarrow$ Verified on IAP. We compare the size of XIVE against that of other systems with similar objectives in Table 1. All are discussed in §5. XIVE is clearly much smaller than Patagonix, due to the fact that Patagonix is incorporated into the full-featured Xen hypervisor. Like XIVE, SecVisor was developed with the specific objective of minimizing its size, so it is much closer. To be fair, we calculated the code size of SecVisor from the breakdown of code they provided, excluding their SHA-1 and module relocation implementations since XIVE does not contain analogous software functionality. SecVisor must use page tables to detect the execution of unverified code and to protect itself, which introduces additional complexity compared to XIVE. We thank the authors of [4] for furnishing us with the relevant line count in Table 1, which includes comments and debugging code, and could perhaps be reduced by future optimizations.

System	Lines of Code
XIVE	932
Patagonix	3544 + ∼230K (Xen)
SecVisor	2682
HyperSentry	∼3400

Table 1: Comparison of the TCB sizes of various systems.

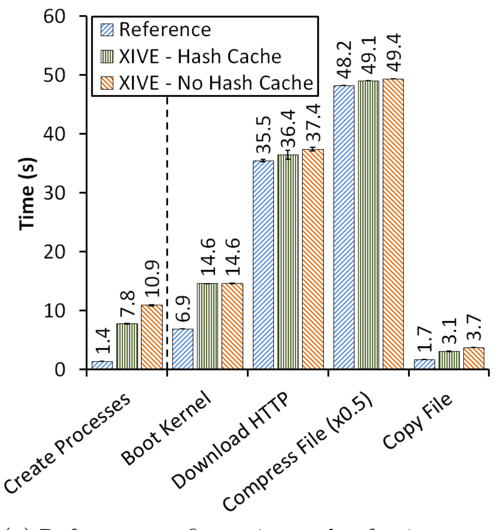
Performance Methodology. We evaluated the performance implications of XIVE using a series of benchmarks. The whole series of tests was run in sequence ten times for each processor configuration. The version of Linux running on the reference system retains a network driver receive buffer handling adaptation that introduces substantial overhead in network processing, since we want to highlight the overhead introduced by XIVE's network traffic itself. The driver checks the whole DMA buffer for received messages during each poll operation, since XIVE can introduce "holes" in the buffer by removing messages that it receives. It may be possible to optimize the network driver in the future to reduce its overhead.

Figure 2a shows results from testing XIVE in two configurations. The one labeled "Hash Cache" includes the full functionality of the XIVE kernel as described in §2.4. The one labeled "No Hash Cache" disables the hash caching functionality, since it is not obvious a priori which configuration imposes less overhead.

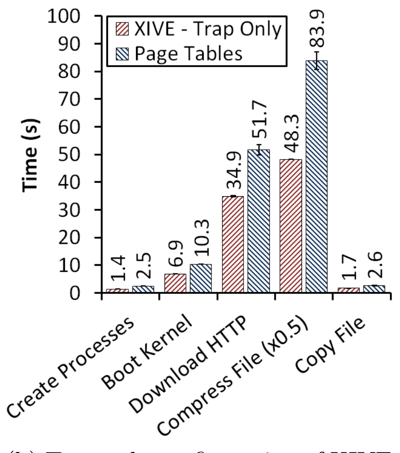
Additionally, to demonstrate the overhead inherent in software-based approaches, we modified the Linux kernel to use page table manipulations to trap attempts to execute unverified code. The kernel does not actually hash or verify the code, so most of the overhead is generated by the page table manipulations and associated traps themselves. We compared the results of benchmarks running that configuration on the reference hardware, labeled "Page Tables," against a third configuration of XIVE that does not perform any hashing or network communication, labeled "Trap Only," in Fig. 2b.

Each of the configurations just discussed was used to run a series of five tests: *1) Create Processes:* A microbenchmark that demonstrates process creation overhead in an adverse case. It executes the `ls` command in an empty directory 10 times in succession. Since `ls` is a lightweight command that performs little work in this case, it demonstrates the time that XIVE requires to verify code during process creation and destruction. *2) Boot Kernel:* This tests the time it takes the Linux kernel to boot. We considered the kernel to be fully booted when we detected that the `init` process had been launched. *3) Download HTTP:* This demonstrates that XIVE is capable of sharing the Ethernet interface with the target OS. The end node downloaded a 2MiB file containing random data using the `wget` command from the LigHTTPD server running on the approver machine. *4) Compress File:* This demonstrates the effect of XIVE on a computationally-expensive process by compressing the previously-downloaded file using GZip. GZip ordinarily involves a mixture of IO and computational operations, but the entire filesystem of our prototype is hosted in RAM, so IO is relatively fast. This fact can also be derived by noting the time difference between this test and the next one showing the cost to copy the same file (*5: Copy File*). We scaled all results from the compression test by a factor of 0.5 to prevent them from dominating the chart.

The FPGA was directly connected with an Ethernet cable to the machine running the approver. That machine was a Thinkpad T61 with a 2GHz Core 2 Duo processor and 2GiB of RAM running Ubuntu 10.10 64-bit desktop edition.



(a) Reference configuration and enforcing configurations of XIVE. The test to the left of the dashed line is a microbenchmark, designed to demonstrate process creation and destruction overhead in an adverse case.



(b) Trap-only configuration of XIVE and page table-based software implementation.

Fig. 2: Mean time required to perform benchmarks, including bars to show the standard error.

Performance Results. In general, the benchmark results in Fig. 2a demonstrate that XIVE imposes low overhead for important types of tasks. However, we present a detailed explanation of the "Create Processes" microbenchmark results as well as those of the "Boot Kernel" benchmark below.

We hypothesized that the repeated verification of shared kernel code and userspace libraries was partially responsible for the order of magnitude performance degradation observed between the reference and the "No Hash Cache" configuration, which is what prompted us to develop the hash caching scheme. It is apparent from the results generated by the "Hash Cache" configuration that caching hashes for resident pages of memory dramatically reduces process creation and destruction overhead. It is also apparent that the overall effect of hash caching on the large-scale benchmarks is often positive.

Booting the kernel involves verifying a large quantity of code, including several verifications of the entire 16MiB system image prior to setting up fine-grained page tables, so boot time suffers a substantial slowdown. It is possible to modify the non-MMU V bits to operate with a finer granularity or to use a more sophisticated mapping structure to reduce this overhead, but that increases hardware complexity and seems unwarranted given the fact that this is a one-time cost per system reset.

We used the "Hash Cache" configuration to determine that XIVE generated an average of 3.7MiB of verification-related network traffic with a standard error of 7KiB as a result of running each series of tests and associated administrative commands.

5 Related Work

SecVisor seeks to ensure that all kernel code ever executed is approved according to a user-defined policy [18]. The prototype uses a whitelist of hashes as the policy. SecVisor uses a variety of mechanisms to enforce the policy, all based on an underlying hypervisor. XIVE monitors all code executed on the target system, including userspace.

Several projects have used Xen to isolate an integrity service from a target VM that is monitored. Lares allows the integrity service to insert hooks at arbitrary locations within the monitored VM that transfer control to the integrity service. *Hypervisor-Based Integrity Measurement Agent (HIMA)* enforces the integrity of user programs running in target VMs by intercepting security-relevant events such as system calls leading to process creation in the target and by permitting only measured pages to execute [3]. Patagonix also ensures that only measured pages can execute [13]. These approaches all have large TCBs, due to their reliance on Xen. Lares and HIMA also have the challenge of determining the proper locations for hooks and the proper types of intercepts, respectively, to achieve comprehensive protection.

HyperSentry and HyperCheck both use SMM to isolate an integrity service while it monitors the integrity of a hypervisor [4, 20]. HyperCheck also offloads a significant amount of functionality to a DMA-capable PCI *Network Interface*

Card (NIC) [20]. Both exhibit *Time-Of-Check-To-Time-Of-Use* vulnerabilities, due to their periodic nature, and also depend on comprehensively identifying security-critical code and structures. However, they do have the advantage of measuring program data as well as instructions.

ARM TrustZone includes a collection of hardware isolation features that could be used to protect an integrity kernel [2]. However, TrustZone does not include support for directly detecting the execution of unverified code. Intel Trusted Execution Technology and AMD Secure Virtual Machine have similar characteristics [1, 10].

TrustVisor creates small VMs that isolate individual functions from a larger overall system and persists their state using TPM-based sealed storage [14]. XIVE monitors and controls the configuration of the whole system, which is largely an orthogonal concern.

The Cell Broadband Engine Isolation Loader permits signed and encrypted applications to be loaded into the Synergistic Processing Elements in the Cell processor [15]. Unlike XIVE, this architecture does not perform any ongoing monitoring of the executed code.

6 Conclusion

IAP is a processor technology that is specifically designed to efficiently support a variety of integrity kernels. It provides high performance, hardware-enforced isolation, high compatibility with target systems and flexible invocation options to ensure visibility into the target system. We demonstrated the utility of IAP by developing XIVE, a code integrity enforcement service with a client component that fits entirely within IAP's protected space, containing 859 instructions. XIVE verifies all the code that ever executes on the target system against a network-hosted whitelist, even in the presence of DMA-capable attackers.

Acknowledgments. This work was supported in part by DOE DE-OE0000097, NSF CNS 05-24695, HHS 90TR0003-01, NSF CNS 09-64392, NSF CNS 09-17218, NSF CNS 07-16626, NSF CNS 07-16421, and grants from the MacArthur Foundation, and Lockheed Martin Corporation. The views expressed are those of the authors only. We thank Samuel T. King and the anonymous reviewers for their helpful feedback. We measured lines of code using David A. Wheeler's 'SLOC-Count'.

References

1. Advanced Micro Devices: AMD64 architecture programmers manual, volume 2: System programming. Publication Number: 24593 (Jun 2010)
2. ARM Limited: ARM security technology—Building a secure system using Trust-Zone technology. PRD29-GENC-009492C (Apr 2009)

3. Azab, A.M., Ning, P., Sezer, E.C., Zhang, X.: HIMA: A hypervisor-based integrity measurement agent. In: Proceedings of the 25th Annual Computer Security Applications Conference. pp. 461–470. ACSAC '09, Honolulu, HI, USA (Dec 2009)
4. Azab, A.M., Ning, P., Wang, Z., Jiang, X., Zhang, X., Skalsky, N.C.: HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In: Proceedings of the 17th ACM conference on Computer and Communications Security. pp. 38–49. CCS '10, Chicago, IL, USA (Oct 2010)
5. Bellare, M., Rogaway, P., Wagner, D.: The EAX mode of operation. In: Proceedings of the 11th IACR Workshop on Fast Software Encryption. pp. 389–407. FSE '04, Delhi, India (Feb 2004)
6. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a board range of memory error exploits. In: Proceedings of the 12th USENIX Security Symposium. Security '03, Washington, DC, USA (Aug 2003)
7. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM conference on Computer and Communications Security. pp. 27–38. CCS '08, Alexandria, VA, USA (Oct 2008)
8. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198–208 (Mar 1983)
9. Duflot, L., Levillain, O., Morin, B., Grumelard, O.: Getting into the SMRAM: SMM reloaded. In: CanSecWest '09. Vancouver, Canada (Mar 2009)
10. Intel: Intel trusted execution technology software development guide. Document Number: 315168-006 (Dec 2009)
11. International Business Machines: IBM X-Force 2010 mid-year trend and risk report. `http://www.ibm.com/services/us/iss/xforce/trendreports/` (Aug 2010)
12. LeMay, M., Gunter, C.A.: Cumulative Attestation Kernels for Embedded Systems. In: Proceedings of the 14th European Symposium on Research in Computer Security. pp. 655–670. ESORICS '09, Saint Malo, France (Sep 2009)
13. Litty, L., Lagar-Cavilla, H.A., Lie, D.: Hypervisor support for identifying covertly executing binaries. In: Proceedings of the 17th USENIX Security Symposium. pp. 243–258. Security '08, San Jose, CA, USA (Jul 2008)
14. McCune, J.M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V., Perrig, A.: TrustVisor: Efficient TCB reduction and attestation. In: Proceedings of the 31st IEEE Symposium on Security and Privacy. pp. 143–158. Oakland, CA, USA (May 2010)
15. Murase, M., Shimizu, K., Plouffe, W., Sakamoto, M.: Effective implementation of the cell broadband engine(TM) isolation loader. In: Proceedings of the 16th ACM conference on Computer and Communications Security. pp. 303–313. CCS '09, Chicago, IL, USA (Nov 2009)
16. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium. Security '04, San Diego, CA, USA (Aug 2004)
17. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: Secure code update by attestation in sensor networks. In: Proceedings of the 5th ACM workshop on Wireless Security. pp. 85–94. WiSe '06, Los Angeles, CA, USA (Sep 2006)
18. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of the 21st ACM SIGOPS symposium on Operating Systems Principles. pp. 335–350. SOSP '07, Stevenson, WA, USA (Oct 2007)
19. SHA-3 proposal BLAKE: `http://131002.net/blake/`

20. Wang, J., Stavrou, A., Ghosh, A.: HyperCheck: A hardware-assisted integrity monitor. In: Proceedings of the 13th international symposium on Recent Advances in Intrusion Detection. pp. 158–177. RAID '10, Ottawa, ON, CA (Sep 2010)