HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity

Zhi Wang
Department of Computer Science
North Carolina State University
zhi_wang@ncsu.edu

Xuxian Jiang
Department of Computer Science
North Carolina State University
jiang@cs.ncsu.edu

***Abstract*— Virtualization is being widely adopted in today's computing systems. Its unique security advantages in isolating and introspecting commodity OSes as virtual machines (VMs) have enabled a wide spectrum of applications. However, a common, fundamental assumption is the presence of a trustworthy hypervisor. Unfortunately, the large code base of commodity hypervisors and recent successful hypervisor attacks (e.g., VM escape) seriously question the validity of this assumption.**

In this paper, we present HyperSafe, a lightweight approach that endows existing Type-I bare-metal hypervisors with a unique self-protection capability to provide lifetime control-flow integrity. Specifically, we propose two key techniques. The first one – *non-bypassable memory lockdown* – reliably protects the hypervisor's code and static data from being compromised even in the presence of exploitable memory corruption bugs (e.g., buffer overflows), therefore successfully providing hypervisor code integrity. The second one – *restricted pointer indexing* – introduces one layer of indirection to convert the control data into pointer indexes. These pointer indexes are restricted such that the corresponding call/return targets strictly follow the hypervisor control flow graph, hence expanding protection to control-flow integrity. We have built a prototype and used it to protect two open-source Type-I hypervisors: BitVisor and Xen. The experimental results with synthetic hypervisor exploits and benchmarking programs show HyperSafe can reliably enable the hypervisor self-protection and provide the integrity guarantee with a small performance overhead.

I. Introduction

Recent years have witnessed the wide adoption of virtualization in today's computing systems. The unique security advantages from virtualization, especially in isolating and introspecting commodity OSes as virtual machines (VMs), have prompted a wave of research [18], [23], [28], [34], [36], [37], [40], [44], [53], [59]. For example, Livewire [18] pioneers the concept of VM introspection and applies it for system monitoring and malware detection. SecVisor [40], NICKLE [36], VMwatcher [23], Lares [34], HookSafe [53], and SIM [44] leverage virtualization to protect the guest OS kernel integrity or enable reliable monitoring of OS kernel behavior. Most recently, a number of virtualization-based system debugging and analysis tools such as K-Tracer [28], PoKeR [37] and AfterSight [12] have been developed to examine system anomalies and study kernel-mode malware, which is considered difficult or in some situations impossible to do with conventional approaches.

One fundamental assumption shared by all these research efforts is the need for a trustworthy hypervisor (or virtual machine monitor - VMM). A typical supporting argument is that the hypervisor has a code base that is much smaller than conventional OSes and thus can be better scrutinized to remove software bugs. Unfortunately, contemporary hypervisors such as Xen [5] and VMware [52] still have a large, complex code base (e.g., Xen 3.4.1 contains ∼230K source lines of code or SLOC). A recent study of the National Vulnerability Database [33] indicates that in the last three years, there were 26 security vulnerabilities identified in Xen, and 18 in VMware ESX. Some of these vulnerabilities can be directly exploited to execute arbitrary code in the hypervisor. Furthermore, successful VM escape attacks [14], [57] as well as the emerging hypervisor rootkits [7], [24] greatly exacerbate the current situation. In light of these attacks, there is a pressing need to investigate effective ways to secure the hypervisor [38].

One natural but challenging approach is to formally verify that the hypervisor is secure. For example, the L4.verified [27] project aims to guarantee the functional correctness of a micro-kernel implementation, i.e., seL4 [26], by formally proving that the C code implementation (with ∼8700 SLOC) correctly and precisely follows the abstract specification and contains nothing more. This is very helpful as it can lead to strong security guarantees, especially in proving the absence of certain types of software bugs (e.g., buffer overflows and null pointer dereferences). However, to perform a formal proof, it imposes several stringent requirements on the micro-kernel design and implementation. For example, the kernel should run with interrupts mostly disabled and no address-of operator (&) and function calls through function pointers will be allowed. Also, the memory management component is moved out of the kernel space and exempted from being formally proved. Further, besides its inherent scalability constraint, we also notice that the proven functional correctness from a manually-specified specification does not necessarily equal the actual safety properties of the system. As a result, though it is an attractive approach,

significant efforts are still needed to make it suitable for commodity hypervisors as their designs are not constrained by these restrictions.

From another perspective, we can tackle this hard problem by guaranteeing runtime hypervisor integrity despite the presence of exploitable software bugs. Common wisdom holds that to secure a running application, one runs monitoring software a layer below the application. However, this is not applicable here, simply because the hypervisor already runs at the lowest level of the software stack. It may be argued that a nested hypervisor can be developed to run underneath and protect another hypervisor running above. However, a fundamental question of the same nature still remains: "how to protect the hypervisor running at the lowest-level?"

Existing hardware-based technologies including TPM [48] and measured late launch [21] are capable of effectively establishing static/dynamic root of trust by guaranteeing the loading of a hypervisor in a trustworthy manner. In other words, they can guarantee the load-time integrity of the hypervisor. However, the main challenge is *how to maintain the same level of integrity continuously throughout the lifetime of the hypervisor.* Due to the fact that we cannot rule out the presence of software vulnerabilities in the hypervisor, we have to address the threat that after the hypervisor is securely loaded, these vulnerabilities may be immediately exploited to sabotage its integrity.

In this paper, we present the design, implementation, and evaluation of HyperSafe, a system that reliably establishes the continuous integrity of the lowest-level software on a system, i.e., the hypervisor. Specifically, continuous integrity in this paper is enforced in the form of lifetime control-flow integrity [1]. Our system is lightweight and can be integrated into commodity hypervisors[1] without requiring specialized hardware support. And unlike the previously mentioned common wisdom, even though HyperSafe is a natural part of the hypervisor, it preserves via a self-protection mechanism the lifetime hypervisor integrity.

In particular, HyperSafe implements two key techniques: The first one is *non-bypassable memory lockdown*, which essentially serves as the cornerstone for the entire scheme and enables the unique hypervisor self-protection. Specifically, once a memory page is locked down, this technique guarantees that the page needs to be unlocked first – even for legitimate hypervisor code – in order to modify the page. And by design, the unlocking logic will simply disallow any attempts that will either modify existing hypervisor code or bring external (malicious) code for execution in the hypervisor space. In other words, this technique locks down those write-protected memory pages (containing hypervisor code and read-only data) as well as their attributes (in the page tables) and prevents them from being changed at runtime, thus effectively achieving hypervisor code integrity. We highlight that the enforcement cannot be bypassed even in the presence of potentially exploitable memory corruption bugs such as buffer overflows.

The second key technique is *restricted pointer indexing*, which essentially leverages the memory lockdown technique to expand the protection coverage from hypervisor code to control data. Notice that when used in related control transfer instructions (e.g., *call/jmp/ret*), the control data can directly impact the control-flow of hypervisor execution. Their security implications become evident, especially with the recent exposure of *return-oriented programming* [20], [43]. Unfortunately, we cannot directly apply the memory lockdown technique to protect all the control data, as some of them (e.g., return addresses in the stack) will be dynamically generated. To address that, we observe the potential control flow always follows the control-flow graph, which can be predetermined ahead of time. With that, we can convert the control data (also called *pointers* in this paper) into pointer indexes and restrict them to be conformant to the control-flow graph. In other words, we can pre-compute possible control flow targets, save them in the target tables, and restrict the accesses from pointer indexes to them. Since these target tables are static, we can directly leverage the memory lockdown technique to protect them. Consequently, the protection of the hypervisor integrity is expanded from the code to the control data for control-flow integrity.

To the best of our knowledge, HyperSafe is the first system that is capable of providing hypervisor control-flow integrity. To validate our approach, we have implemented a proof-of-concept prototype and applied it to the protection of two open source Type-I hypervisors, i.e., BitVisor [46] and Xen [5]. Specifically, the first key technique is implemented by directly modifying the hypervisor source code while the second key technique is implemented as a compiler extension to the re-targetable LLVM framework [30], which is thus hypervisor-transparent. As a result, the BitVisor/HyperSafe prototype is a full implementation with both key techniques. For the Xen port, since the current LLVM release does not support compiling Xen yet, our current prototype only enables the non-bypassable memory lockdown, which still guarantees the nontrivial code integrity of Xen. Our prototyping experience indicates that HyperSafe's code size is small and its integration with commodity hypervisors is straightforward. Evaluation with synthetic hypervisor attacks as well as a number of performance micro-benchmarks and user applications show that the integrity protection can be effectively enabled with less than 5% performance overhead.

The rest of the paper is structured as follows. We first show the overall system design, discuss the threat model, and present the two key techniques in Section II. We show implementation details in Section III and present our

[1]Considering the various flavors of hypervisor implementations (Section II), we focus on those Type-I hypervisors. Some examples of these hypervisors are Xen [5], VMware ESX [52], and BitVisor [46].

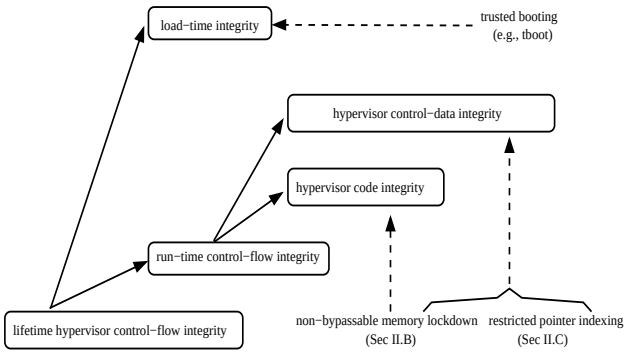


Figure 1. A break-down of hypervisor integrity guarantees and the corresponding key techniques in HyperSafe

evaluation results in Section IV. We discuss the support of Type-II hypervisors as well as possible limitations and improvements in Section V. After that, we describe related work in Section VI and conclude our paper in Section VII.

II. HyperSafe Design

A. Goals and Assumptions

In order to provide lifetime hypervisor control-flow integrity, we have three main design goals. *First*, the proposed techniques should enable the self-protection of commodity hypervisors. As the name indicates, self-protection may not introduce new lower level mechanisms. Further, the self-protection mechanism needs to be *reliable* in the presence of exploitable memory corruption bugs (e.g., buffer overflows or format string bugs) and *effective* in proactively preventing attacks from gaining execution control over the hypervisor.

Second, the proposed techniques should not require restructuring or impacting the original hypervisor design while still providing the desired integrity guarantee. In other words, the proposed techniques need to be generic and amenable to commodity hypervisors without limiting the design choices or imposing implementation restrictions (e.g., in disabling certain programming language features). We may need to tolerate some minor modifications to commodity hypervisors, but the modifications should be minimal. Also, based on the traditional classification between Type-I bare-metal and Type-II hosted hypervisors and the fact that Type-II hypervisors require a hosted OS kernel, our focus in this paper is the support of Type-I hypervisors.

Third, the proposed techniques can be efficiently implemented on commodity hardware, i.e., without relying on sophisticated hardware support to achieve the integrity guarantee or obtain reasonable performance for deployment. Given this requirement, the challenge is to ensure that the proposed techniques can be implemented on top of commodity hardware, have a small footprint, and remain lightweight with respect to performance impact.

Threat model and system assumption In this work, we assume an adversary model where attackers are able to exploit software vulnerabilities in an attempt to overwrite any location in memory. However, to successfully launch an attack, attackers will have to either inject and execute their own code or leverage and misuse existing code. Note this represents a powerful adversary model as attackers can attempt to inject code, modify existing code, and exercise more sophisticated attacks such as return-oriented ones [20].

In the meantime, we do assume trustworthy hardware, especially the TPM [48]-assisted static/dynamic root of trust [49], which can be leveraged to guarantee load-time hypervisor integrity. Due to the unsafe programming language used in the implementation, we do assume the presence of vulnerabilities in the hypervisor. However, the attackers are restricted in their attempts to subvert the hypervisor integrity by only exploiting these vulnerabilities, not by out-of-band attacks (e.g., TLB cache or SMM exploitation [55], [56]) or layer-below attacks (including physical-level attacks). Malicious DMAs [57] are not considered as they can be readily blocked with hardware-based IOMMUs [21]. Notice that our attack model is similar to that used in SecVisor [40]; however, one key distinction is that HyperSafe is designed to protect the hypervisor itself while SecVisor aims to protect the guest kernel code integrity using a small *trusted* hypervisor.

Based on this threat model, we propose two key techniques, i.e., *non-bypassable memory lockdown* and *restricted pointer indexing*, to enable the self-protection of commodity hypervisors. Figure 1 shows a break-down of the required hypervisor integrity guarantees as well as the corresponding key techniques we propose to achieve them. Next, we will describe in detail these two key techniques.

B. Key Technique I: Non-Bypassable Memory Lockdown

As mentioned earlier, this technique serves as the cornerstone for the proposed hypervisor integrity protection. In the following, we first give a brief overview of the available memory protection mechanisms in the Intel *x86* architecture on which our system is developed. In essence, the *x86* architecture supports two types of memory protection: *segmentation* and *paging*. With the introduction of the new 64-bit mode of the *x86* processor, segmentation has been mostly disabled in favor of paging.[2] Because of that, our discussion will be focusing on paging and our system relies on paging-based memory protection.

Specifically, the paging-based memory protection divides the virtual address space into pages, and physical memory into frames of the same size. The translation from the virtual page to the physical frame is facilitated by the page tables. Each page table has a number of page table entries (512 in $x86\_64$). Each entry contains certain bits to specify the corresponding page protection attributes, such as whether the page is writable (the *R/W* bit), executable (the *NX* bit), or

[2]Some specific segments such as FS and GS may still be retained to facilitate the addressing of local data or certain OS data structures.

requires privileged access (the *U/S* bit). Different from the virtual page's privilege levels, the CPU has four privilege levels (or rings) from 0 to 3 with 0 being the highest privilege. The code running in privilege level 3 can only access user pages while the code running in privilege levels 0, 1 and 2 is considered to be supervisor code and can access both user pages and supervisor pages.

With these protection attributes, paging-based memory protection allows for flexible customization to each and every individual page. For example, one common usage of these attributes in commodity OS kernels (e.g., Windows, Linux, and OpenBSD) is to write-protect their code and read-only data. Another similar one is to establish the W⊕X property of the OS kernel to ensure its code integrity as demonstrated in a few recent systems [36], [40].

Similarly, we are motivated to enforce the W⊕X for hypervisor integrity protection. However, there are several notable pitfalls. First, for historical reasons [36], commodity OS kernels may allow the presence of mixed memory pages that contain both code and data. Certainly, the presence of such pages *directly* violates the $W \oplus X$ property and should be avoided in the hypervisor. Second, for performance and efficient resource sharing purposes, existing OS kernels typically allow the mapping of several virtual pages to the same physical frame and different virtual pages may possibly have conflicting protection attributes. Such double mapping *indirectly* breaks the W⊕X property and should not be allowed in the hypervisor either. Third, most importantly, the W⊕X-based integrity enforcement largely relies on the integrity of page tables. For a write-protected page to be modifiable, the corresponding page table entries will need to set in a way to allow it. Unfortunately, in current hypervisors (e.g., Xen and KVM) and OS kernels (e.g., Windows and Linux), their page tables are all *writable*! This implies that even if a hypervisor ideally sets these memory protection attributes, the enforcement can be easily bypassed since the page tables are writable. Our experience indicates that the ability to modify even one bit in a page table entry could well be enough to subvert the entire protection.

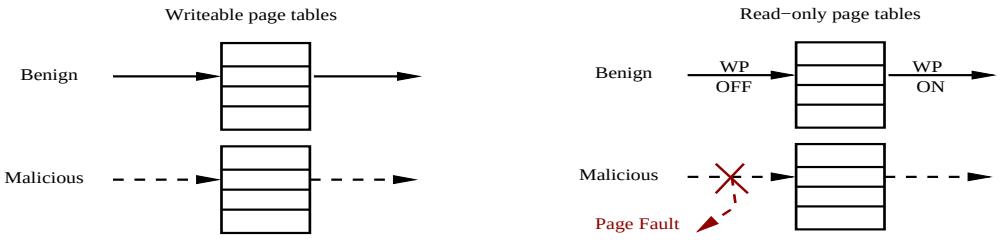
From another perspective, if we assume the proper initialization of the hypervisor page tables (i.e., no mixed pages, no double mapping, and a correct W⊕X setup for each memory page), the attacker will be forced to first manipulate the page tables in order to bypass the W⊕X protection. This observation motivates us to also write-protect the page tables. By doing so, we can ensure that *no code including legitimate code will be able to modify the write-protected hypervisor code (and related control data – Section II-C)*. As mentioned earlier, in order to proceed with any modification, the page tables need to be changed to allow it. But the write-protection of page tables disallows such a change. Consequently, any write attempt to them (including the page tables) will be hardware-trapped into the page fault handler, which, as a part of legitimate code, is unable to modify them either. This is a strong guarantee that serves as the basis to establish and sustain hypervisor runtime integrity.

The above strong guarantee is desirable for hypervisor protection as it can effectively prevent malicious updates to page tables. Unfortunately it will also trap and block all benign updates. Again, the reason is that once paging is enabled, the hypervisor can only access its memory through virtual addresses, which will be translated and subjected to protection checks by the page tables.[3] As a result, the creation of read-only page tables immediately leads to an unsolvable paradox: read-only page tables can detect and deny any malicious manipulation but they also make the benign changes impossible.

To accommodate benign page table updates, we need to design a secure way to temporarily bypass the enforcement without being misused (e.g., by return-oriented attacks). This is how our technique – non-bypassable memory lockdown – comes into play. Specifically, our technique uses a hardware feature called the WP bit (i.e., the Write Protect bit in the machine control register CR0 [58]), which has existed in all *x86* CPUs since the Intel Pentium. The WP bit controls how the supervisor code interacts with the write protection bits in page tables: If the WP bit is turned off, the supervisor code has unfettered access to any virtual memory (i.e., the write-protection is ignored). Otherwise, the write protection attributes in page table entries will decide whether the supervisor code can write to the memory page or not. Note the WP bit was originally introduced to facilitate the Copy-On-Write (COW) implementation of forking a new process. More specifically, in Linux, when a process forks, memory pages are COW-shared (or marked as read-only) between parent and child processes. Therefore, any write to a COW-shared page leads to the creation of a new copy of the page and the sharing can then be removed. As a result, OS kernel can simply set the WP bit to trap its own writes to these pages, which greatly simplifies the COW design and implementation. (Otherwise, OS kernel must check for COW every time it writes to user space.)

With that, we can initially mark the page tables read-only and turn on the WP bit to lock down any page table updates, regardless of their intent being benign or malicious. To allow benign ones to proceed, we can instead escort them by temporarily clearing the WP bit right before each update and re-enabling the bit right after. Naturally, the entire escort operation needs to be atomic (e.g., with interrupts disabled). Otherwise the attackers may potentially interrupt the operation and leave the WP protection off. Within the escort operation, HyperSafe can further validate that the new page table entries conform to the security policy, which can be specified by the hypervisor developer. In our

[3]Note that although the CPU's hardware translates virtual addresses with page tables, the hardware's accesses are not translated as the CPU uses physical addresses directly. Therefore the CPU has no trouble at all to read and update them, e.g., to set A (accessed) and D (dirty) bits.



(a) Traditional page table updates (b) New page table updates

Figure 2. Traditional page table updates vs. new page table updates in HyperSafe (Note the *WP* bit is *ON* by default)

implementation, we enforce a simple invariant that denies page table updates that attempt to change the protection attributes of hypervisor's code and data or introduce a double mapping. Though it may appear that this security check is redundant once the hypervisor's control-flow integrity is protected, we show that this is not the case in Section IV-A. Figure 2 shows the comparison between traditional page table updates and the new page table updates in HyperSafe.

It is worth mentioning that to protect the hypervisor page tables, we also need to protect related machine control registers and data structures. For example, the hypervisor's page table base address is contained in $CR3$, which should not change after the initialization. The same also applies for the entries in the GDT (Global Descriptor Table) and the IDT (Interrupt Descriptor Table). In addition, the hypervisor virtualizes the guest's memory by using either shadow page tables (SPTs) or nested page tables (NPTs). The updates to them need to be protected in a similar way to prevent the attacker from gaining the control over the hypervisor's memory (e.g., by mapping it to a compromised guest).

To summarize, our technique effectively locks down hypervisor memory pages to strictly specify and manage memory protection attributes. Most importantly, the enforcement of these memory attributes such as W$\oplus$X is non-bypassable (Section IV-A). As a result, we can reliably provide hypervisor code integrity. Next, we will present another key technique that essentially expands the protection coverage to control-data and enables control-flow integrity.

C. Key Technique II: Restricted Pointer Indexing

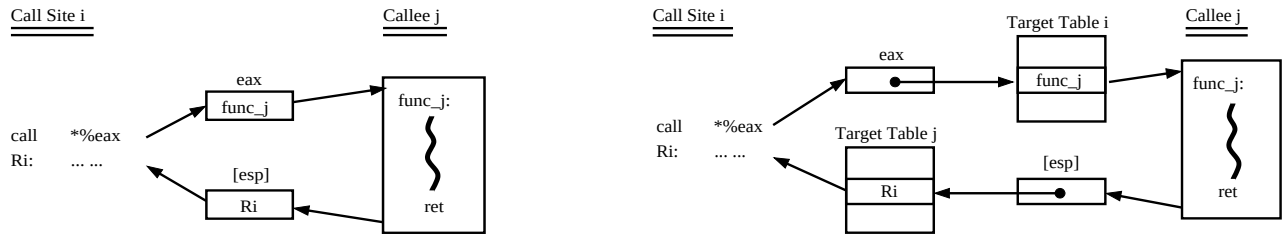
Control flow integrity (CFI) is a powerful security measure, which strictly dictates the software's runtime execution paths. If the software's runtime paths follow the statically determined control flow graph, attackers can be prevented from arbitrarily controlling the execution flow of the system. Based on how control is transferred, there are two types of control flow transfer instructions: direct and indirect. A *direct control transfer* is initiated by a direct function call where the destination is encoded in the machine code in the form of an absolute address or a relative offset. Accordingly, control-flow integrity from direct control transfers is maintained as long as the code integrity is guaranteed.

An *indirect control transfer*, which is our main focus, can be caused by two sets of instructions: indirect *call/jmp* instructions (where the destinations may be specified in registers or memory) and the *ret* instructions. Each *ret* has an implicit destination on the top of the current stack. Correspondingly, there are two types of control data: function pointers and return addresses. For simplicity, we also call them *pointers*. Due to the dependence of indirect control flow on these pointers, we need to protect their integrity to preserve indirect control flow integrity.

Unfortunately, we face three main challenges: *First*, control data can be widely scattered in memory and can coexist together with other dynamic data on the same pages [53]. Naive page level protection will likely lead to huge performance overhead. *Second*, some control data can be dynamically generated and thus their locations cannot be determined a priori. A representative example is the return address. This implies any protection scheme that requires their locations to be static will fail. *Third*, some control data such as return addresses can be updated at a high frequency. This invalidates any approach that requires write-protecting frequently updated control data. Our experience with a local build of the Xen hypervisor (version $3.4.1$) for the *x86-64* architecture indicates that there is a *call* instruction on average for every 23 machine instructions in the binary and a *ret* instruction for every 79 instructions. At runtime, each *call* instruction will push a return address onto the stack, which will then be popped off by the corresponding *ret* instruction.

From another perspective, we notice that though the control data may be dynamically generated or frequently updated, their contents always fall in a data set that can be determined offline. As such, we can aggregate them into individual target tables and, by introducing one layer of indirection, replace each control data with a restricted index to the target table (hence the name *restricted pointer indexing*). More specifically, the target table contains all the legitimate destinations for an indirect control flow instruction allowed by the hypervisor program's control flow graph (CFG). For each indirect *call/jmp*, its table contains the function entry points it may enter. Similarly, the target table for a *ret* includes all the return addresses it may return to.

Based on the target tables, HyperSafe can essentially replace all the runtime control data in the hypervisor program with their indexes in the target tables. To perform a control transfer, the indirect control flow instruction will be



(a) Traditional indirection call (b) New indirection call

Figure 3. Traditional indirect call vs. new indirect call in HyperSafe (Note *Ri* is the return address of the indirect *call*)

instrumented to convert the index back to the destination address (e.g., by looking up the index in the table). For that, we need to take the following two steps:

First, the instructions that introduce the control data into the hypervisor program must be converted to use the indexes instead. For simplicity, we call these instructions *source instructions*. The source instruction for a return address is the related *call* that pushes the return address onto the stack. As a result, the call instruction will be instrumented into two instructions: one *push*es the index onto the stack and another *jmp*s to the function entry point. For an indirect *call*, its source instruction is an earlier instruction that loads the function address to the register or memory. Unlike the return address case, the function pointer can possibly appear in the data section (e.g., as a member of an initialized global object or variable). As a result, we can leverage the compiler to identify and convert them.

Second, the instructions that consume the control data from the hypervisor program must be converted to translate the indexes back to their destination addresses. Similarly, we call these instructions *sink instructions*. Return addresses will be used by the *ret* instructions while function pointers will be consumed by indirect *call/jmp* instructions. During instrumentation, a *ret* will be converted to a sequence of instructions to pop the index off the stack, convert it into the return address, and then return to it. An indirect *call/jmp* will be converted to use the index to locate the function entry point and then continue execution there.

Based on the above instrumentation, an indirect *call* acts as a sink instruction for the consumed function pointer and a source instruction for the dynamically-pushed return address. Therefore, it will be instrumented twice. There may also exist other instructions that access the control data but are not the source and sink instructions. Among them, some instructions can be left intact if the contents of the control data are not explicitly examined by them. One example is the mov instruction that copies the index to and from registers or memory. Instructions that compare two function addresses do not need instrumentation either if we assign the pointer indexes in the order of their addresses. On the other hand, instructions that examine the contents of control data must be expanded to convert indexes into original control data. A general solution is to discover and convert all such instructions, ideally by the compiler. Fortunately, very few instructions will touch return addresses on the stack. If they do, most likely they are implemented in assembly and thus we can instrument them manually. For function pointers, most accessing instructions are mov or cmp. In this case, the contents of the function pointers are not examined and we can safely keep these instructions as is.

In Figure 3, we show the control flow for an instrumented *call/ret* pair in HyperSafe when compared to the original pair. In the figure, the original *call* has been instrumented to fetch the index from eax, convert it to a function entry point by indexing into its target table, and then jump to the function. By substituting indexes for control data, HyperSafe limits the destination of a runtime control transfer to only those explicitly specified in the target table. In other words, indirect instructions can only transfer control to the targets allowed by the CFG. Moreover, because all the destination addresses are known beforehand from the hypervisor program binary, these target tables can be pre-computed offline. At runtime, they are protected by directly applying the memory lockdown technique.

Furthermore, with the help of the target tables, HyperSafe can flexibly control the precision of control-flow integrity. In one extreme case, we can simply use two big tables: one is for all the *ret* instructions (with all valid return addresses) and the other one is for all the indirect $call$ instructions (with all possible indirectly-called functions' entry points). This scheme provides the least precision, resulting in coarse protection: namely a *ret* can return to any valid return address in the hypervisor program; and an indirect *call* can call any indirectly-called function. On the other extreme, each indirect *call* has its own target table, and all *ret* instructions inside the same function share one target table. In other words, each function has a dedicated table for all of its returns. By doing so, we can provide the finest control over what targets indirect instructions can transfer control to. Note that there is no need to use one target table per return instruction since all the *ret* instructions in a function always have the same set of return addresses.

As pointed out in [1], the major factor that impairs the precision of control-flow integrity is the so called *destination equivalence* effect. That is, two destinations are considered to be equivalent if they connect to a common source in the CFG. Further, the equivalence relation is transitive. In Figure 4, we show an example of the destination equivalence

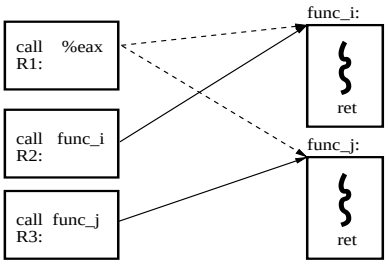


Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect *call* while a solid line stands for a direct *call*)

effect on the ret instructions. In this figure, there are one indirect $call$ instruction and two direct $call$ instructions. The indirect $call$ may invoke both functions $func\_i$ and $func\_j$ while the two direct $calls$ execute $func\_i$ and $func\_j$, respectively. $R1$, $R2$ and $R3$ are the corresponding three return addresses. From the figure, the function $func\_i$ can return to $R1$ and $R2$, and the function $func\_j$ can return to $R1$ and $R3$. Because of the destination equivalence effect, $R1$, $R2$ and $R3$ are all equivalent in this example. More specifically, since $R2$ is equivalent to $R1$ and $R1$ is equivalent to $R3$, based on the transitivity of the equivalence relation, $R2$ is equivalent to $R3$. The destination equivalence effect also indicates that a return address has the same index in each target table that contains it. This is obvious since only one index can be assigned to a specific destination. In our example, $R1$, $R2$ and $R3$ forms one equivalent group, and two ret instructions in $func\_i$ and $func\_j$ can return to them. If one table per function is used to enforce the control-flow integrity, we can use a table "$R1$, $R2$, $error$" for the ret instruction in $func\_i$, and another table "$R1$, $error$, $R3$" for the ret instruction in $func\_j$, where $error$ denotes a special destination to trap an impossible control transfer. Therefore, our one-table-per-function-based control-flow integrity enforcement policy is more precise than the one originally proposed in [1], where $R1$, $R2$ and $R3$ will bear the same label ID and both ret instructions can legitimately transfer control to all of them. In particular, in [1], the function $func\_i$ can legally return to $R3$ and $func\_j$ can legally return to $R2$. In comparison, our scheme can flexibly handle the destination equivalence effect and make these two paths simply impossible in HyperSafe.

III. Implementation

We have implemented a prototype of HyperSafe and applied it to protect two open-source Type-I hypervisors, i.e., BitVisor [46] (with ∼190K SLOC) [4] and Xen [5] (with ∼230K SLOC). In particular, the first technique – non-bypassable memory lockdown – is implemented by directly extending their memory management modules. For the second technique – restricted pointer indexing, we choose to extend the open-source LLVM compiler so that we can enable it by simply re-compiling the hypervisor code with the modified compiler. Our development environment is a standard 64 bit Ubuntu 9.10 desktop. As mentioned earlier, the BitVisor port is a full implementation, while the Xen port only contains the non-bypassable memory lockdown feature, which nevertheless guarantees the nontrivial code integrity of Xen. Meanwhile, our current prototype integrates the trusted booting software, i.e., tboot [49], to protect the load-time integrity. After the hypervisor is successfully loaded, HyperSafe will then ensure its runtime integrity. In the following, we focus on the BitVisor port as an example to present our implementation details.

[4]In our prototype, we disabled the VPN support in BitVisor as it is not relevant.

A. Non-Bypassable Memory Lockdown

The key novelty of our system is the non-bypassable memory lockdown technique for hypervisor integrity protection, achieved purely based on commodity hardware support. Specifically, HyperSafe write-protects the hypervisor's page tables and turns on the WP bit in $CR0$ to initiate the memory lockdown. Our system requires only minimal modifications to the supported hypervisors, therefore satisfying the second design goal (Section II). Specifically, in our BitVisor prototype, we only added or changed 521 lines of C code and 9 lines of assembly code. To avoid potential pitfalls in W⊕X enforcement (Section II), we adjust the link script to align related sections to avoid mixed pages and at runtime disallow double mappings.

In our prototype, we reserved the top $128MB$ physical memory for BitVisor. This memory is mapped $1:1$ to the virtual address $0x40200000$. A $32MB$ memory range, starting at the virtual address $0x40800000$, is reserved as the shared page table pool from which all the hypervisor's page tables are allocated. After secure booting from tboot, the hypervisor properly initializes the page table data structure, turns on the WP protection in the $CR0$ register, and then enables the paging mode. After entering the paging mode, every virtual memory access will be automatically translated through page tables. Because of that, all the page tables have to be accessible and mapped in the hypervisor's virtual address space. In BitVisor, since all the page tables are allocated from and mapped in the page table pool, we simply set the whole page table pool as read-only to lock the page tables. To accommodate benign updates, our system first traverses through the page table hierarchy to locate the affected page table entries, and then escorts their updates to guarantee that existing hypervisor code will not be modified and no external code will be introduced for execution.

After the page tables have been write-protected, any write attempts to modify them at runtime (e.g., either by legitimate hypervisor code or malicious code injected due to a successful exploitation) will be trapped. Inside the page fault handler, we will enforce an unlocking logic that simply preserves the W⊕X property. In the meantime, there also exist a number of legitimate reasons for the hypervisor to

update its page tables without violating the W⊕X property. For example, the hypervisor may need to map part of guest memory pages or device memory for its access. This mapping is typically temporary as it will be removed immediately after the hypervisor has accessed it. For that, instead of triggering a page fault, the hypervisor first turns off the WP protection, updates related page table entries, and turns it back on. The BitVisor implementation provides several helper routines, i.e., *pmap_wr64* and *pmap_wr32*, that are used to update page table entries. Our prototype wraps these routines by adding additional inline assembly code to turn WP protection on and off. To ensure that such escort operations are atomic, our prototype disables the interrupts when an escort operation is in progress. Further, in order to prevent the misuse of these routines, HyperSafe validates whether the change is benign. Our current prototype simply denies any change to the permission of the hypervisor's code and data sections after initial setup. Also, it disallows the double mapping of the hypervisor's code and data sections. This check can be implemented efficiently by verifying the page table update against various address ranges (e.g. physical or virtual address ranges of hypervisor's code and data sections). As discussed in Section IV-A, this check is *not* redundant even in the presence of a control-flow integrity guarantee.

In our prototype, we use the memory lockdown feature to write-protect not only the hypervisor's code, but also its static data. Some examples of this data include the entries in the GDT, the IDT, and various target tables. As mentioned earlier, all these data structures are security-critical and should be write-protected by HyperSafe. For guest page tables, there are two main virtualization modes: shadow page table (SPT) and nested page table (NPT). In the SPT mode, hypervisor "shadows" the guest page tables by maintaining a corresponding copy, i.e., the shadow page tables. The CPU translates the guest virtual address directly to the host physical address using these shadow page tables. The hypervisor also traps updates to the guest page tables and synchronizes shadow page tables with them. In the NPT mode, there are two levels of page tables used by the hardware. The CPU first translates a guest virtual address to a guest physical address with the guest page tables, and then maps that guest physical address to a host physical address based on nested page tables maintained by the hypervisor. Note that the shadow page tables (if running in the SPT mode) or the nested page tables (if running in the NPT mode) need to be protected so that the hypervisor memory will not be accidently mapped and accessible to the guest.

B. Restricted Pointer Indexing

Our second technique replaces all the control data with their indexes and essentially leads to the protection of control data to enforce control-flow integrity. Specifically, it first aggregates all the possible destination addresses (function entry points and return targets) in a few read-only target tables, and then replaces the destination addresses used by the program with their indexes. By doing so, we can guarantee that an indirect control flow transfer instruction that utilizes one of these protected pointers will transfer the control *only* to the addresses specified in the target tables. As discussed earlier, we can also flexibly control the precision by adjusting these target tables to handle the destination equivalence effect. In our prototype, we implemented a scheme that uses one table for each indirect *call/jmp* instruction and one table for each function (applicable for all the *ret* instructions contained in the function). The table per indirect *call/jmp* instruction contains all the function entry points it may call. Similarly, the table per function has all the valid return addresses for this function. As such, the target tables reflect the hypervisor's control flow graph.

Our technique is implemented as a compiler extension to the re-targetable LLVM framework. We choose it because it is a production-quality open source compiler infrastructure that has a modular design and can be flexibly extended. In particular, our extension includes a program analysis and optimization phase that can be applied to the intermediate representation (IR) of a program. In essence, it integrates the supported alias analysis to build and generate the target tables while extending the compiler back-end to instrument instructions for the use of target tables. Note the target tables contain information for both direct and indirect calls. The control flow graph for the direct calls is simple to extract because their targets are already encoded in the instructions. For the control flow graph of indirect calls, we extend the existing alias analysis in LLVM. Specifically, we leverage the data structure analysis implemented in LLVM to identify possible call targets. Note that the data structure analysis is a context-sensitive, field-sensitive unification-based pointer analysis. As a result, it is considered conservative. During our implementation, we found that it is relatively effective to analyze C code. However, it is unable to handle assembly, which is one common limitation of existing alias analysis tools.

In our prototype, we realize that one specific way BitVisor saves and utilizes function pointers foils the data structure analysis. In particular, to facilitate multi-core support, BitVisor keeps a per-cpu data structure and accesses it with the help of the gs segment. (Note gs is one of the two segments that $x86\_64$ keeps to allow for such access.) With that, each processor can set its gs segment's base or gs_base to a location different from other processors. And the hypervisor can conveniently access the per-cpu data with the $gs : offset$ addressing mode, which will be translated by the current CPU into the $gs\_base + offset$. As assembly code is required to load the per-cpu data, the data structure analysis in LLVM is unfortunately unable to uncover all the call targets. To handle that, we manually went through the indirect calls that were not able to be analyzed

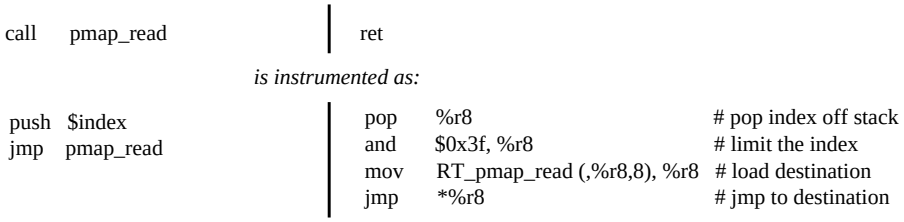


Figure 5. The instrumented *call* to $pmap\_read$ and the corresponding *ret* (note the instrumentation is done in $x86\_64$)

and then leveraged our domain knowledge to resolve them. Our experience shows that there are 126 indirectly called functions and 360 indirect call sites in BitVisor, and the data structure analysis was able to extract indirect call targets for about three fourths of them. Therefore, we had to manually analyze the rest. We point out that most of these manual efforts only need to be done once, though ideally we still need an automated approach. For that, the recent efforts [8] on alias analysis at the assembly level can be naturally integrated and extended.

Once the complete call graph is derived, it is a rather straightforward process to generate the target tables. In our prototype, we use the standard union-find algorithm to locate the equivalent targets and assign the same index to the destinations if they appear in multiple target tables. Also, our prototype assigns $error$ to certain target table entries if they are not reachable in the call graph. In our build of the hypervisor, it turns out that there are 681 target tables for *ret* instructions and 360 target tables for indirect *calls*. If we include direct calls, there are 4100 call sites in total.

After building the control-flow graph and generating the target tables, we extend the compiler back-end to instrument relevant source and sink instructions. In our build environment, the compiler first compiles the C source into assembly code, and then generates the executable binary with GAS, the GNU Assembler. Our prototype extends a LLVM component called AsmPrinter to generate the instrumented assembly code. Specifically, a *call* instruction is instrumented to push the index of its return address to the stack, while a *ret* is instrumented to pop the index off the stack, fetch the actual return address from its target table, and continue execution there.

To better describe the scheme and discuss one possible optimization we identified and implemented in our system, we consider the instrumentation of a direct *call*. In Figure 5, we present the results after instrumenting a direct *call* to the function $pmap\_read$ and the corresponding *ret* in that function. Basically, the instrumented $call$ instruction pushes the index of the return address to the stack followed by a jump to the target function. The ret instruction is rewritten to first pop the index off the stack, then load the actual return address from its target table (RT_pmap_read) and resume execution to it. In this example, RT_pmap_read is defined as a static global variable to contain the return addresses of 58 calls to $pmap\_read$. For alignment purposes, the table actually contains 64 elements and invalid elements are filled in with $error$ – the address of an error reporting routine to trap illegal control flows. In the meantime, it also allows HyperSafe to prevent possible index overflows with one and instruction (the second instruction in the instrumented *ret* as shown in the figure). To reduce the performance overhead, we use a caller-saved register, i.e., $r8$, to facilitate the instrumentation and avoid unnecessary register spilling.

Furthermore, in the case that a target table only contains a single entry, we apply an optimization that avoids the instrumented ret instruction to read the target address from the target table. Specifically, the instrumented *ret* will be a simple *add* that increments the rsp register by 8 bytes to remove the return address index from the stack, followed by another direct *jmp* to the corresponding target. Note this optimization has performance benefits and is made possible because the function is only called from a particular call site. When compared to the unoptimized case (Figure 5), this optimization saves two memory accesses (in *pop* and *mov*, respectively) and replaces the indirect jmp with a direct jmp. In our prototype, we found that 291 out of 681 target tables for return instructions contain only one entry and their optimizations greatly contribute to reducing the performance overhead of the system.

The indirect calls are handled in a similar way. Since the targets for these indirect calls are function entry points, we need to convert the uses of function entry points to their indexes. Accordingly, we convert those source instructions (e.g. mov) that load function addresses into registers to load their indexes instead. Note the locations of these source instructions are easy to identify in the compiler since one of its operands is actually the symbol of the function. After that, we replace the function addresses in data structures with their corresponding indexes. Our system instruments the corresponding sink instructions (at the corresponding indirect call sites) to examine the indexes from their operands and convert them back to the function entry points (i.e., by indexing into the target tables). Similarly, we can also apply the previous optimization if a particular indirect call has only one target in the target table. By doing so, we can replace the indirect call with a direct call and improve performance by avoiding one memory read (of the function pointer). In our prototype, we found 294 out of 360 target tables for indirect calls contain only a single entry and can thus be optimized. Further investigation shows that most of these optimized

indirect calls are due to the specific way taken by BitVisor to support the two existing X86 hardware virtualization architectures, namely Intel VT and AMD SVM. In particular, BitVisor defines a common interface for the support of Intel VT and AMD SVM. The common interface contains a set of function pointers to abstract the difference between them so that the upper layer software can be shielded from low level details. With that, since only one architecture is possible at runtime, we optimized these function pointers at the compiler time according to the CPU used. Besides these source and sink instructions, we do not observe the presence of any other instructions that examine the function pointers' content. In other words, there is not a single arithmetic operation on function pointers.

It is also interesting to mention that in the early development of our second technique, we explored another way to handle ret instructions: shadow stack. Specifically, each $call$ pushes a copy of the return address to the shadow stack and each ret fetches the return addresses from both the shadow stack and the original stack and then compares them to detect any corruption. However, one challenge we realized is how to effectively protect the shadow stack. One feasible approach on the $x86$ architecture is to use segmentation: the shadow stack is kept on an isolated segment from the segments used by the normal code and will be only accessible by instrumented instructions. Unfortunately, segmentation support is largely disabled on the $x86\_64$ platform. Another possibility is to make the shadow stack read-only with the same memory lockdown technique (i.e., by wrapping the $call/ret$ instruction with the code to dynamically switch the WP bit on and off). We have actually implemented this approach but our experiments show that it incurs a performance penalty of more than 300%. This is rather disappointing. The reason is that the return addresses are frequently generated and the instructions to read and write CR0 (to change the WP bit) are more expensive than other regular instructions. After these failed attempts, we eventually ended up with the current scheme that achieves the desired security properties with a small performance overhead (Section IV).

To summarize, our second technique effectively protects the control data and allows for a stronger hypervisor control-flow integrity guarantee from the original code integrity.

IV. Evaluation

In this section, we first analytically examine the security guarantees provided by HyperSafe. Then we present our experiments with synthetic hypervisor exploits and measure the performance overhead.

A. Security Analysis

As mentioned earlier, HyperSafe implements two key techniques: non-bypassable memory lockdown and restricted pointer indexing. The first technique essentially guarantees the hypervisor code integrity and the second technique expands the protection to enforce control-flow integrity. In the following, we systematically examine possible threats to these security guarantees.

To subvert the hypervisor's integrity, an attacker's main goal is to modify existing hypervisor code or introduce and execute its own attack code in the hypervisor space. Since the modifiability of existing hypervisor code and executability of introduced attack code are governed by the hypervisor page tables, the attacker needs to first subvert the page tables. Due to non-bypassable memory lockdown in HyperSafe, these page tables are write-protected with the WP bit on. In the following, we examine two possible attacks to subvert the protected page tables.

Disabling the *WP* bit The first attack aims to turn off the WP bit. Since the attackers are not yet able to inject and execute their own attack code, they must misuse existing hypervisor code. To accommodate benign page table updates, HyperSafe does introduce additional code to temporarily turn off the WP bit. Specifically, to legitimately update a page table, HyperSafe uses an atomic function that disables interrupts and turns off the WP protection before updating the page table. Immediately after the update, the WP protection is turned back on again. With that, in order to disable the WP bit, the attacker must divert the normal execution flow (i.e., by hijacking control data) before the WP bit is turned on again. Based on the control-data protection by the second key technique of HyperSafe, such a diversion attempt is effectively defeated.

Another possible method is to compromise a previously saved runtime context. For example, the hypervisor may save important control registers (such as $CR0$ with the WP bit and $CR3$ as the page table base address) to writable memory and later restore them. The attacker could potentially gain full control of the execution if these saved states can be tampered with. In our prototype, we ensure the correctness of these states before restoring them, i.e., their values are not changed after being initialized.

Subverting the page tables Alternatively, the attackers could subvert the page tables. As before, due to their inability to directly execute their own code, the attackers have to misuse existing code. For that, the attackers may attempt to introduce a double mapping (Section II) to bypass the protection. Specifically, they can change or provide malicious parameters to those normal routines that handle benign page table updates. Based on our current adversary model, this attack is possible (even if we can faithfully enforce control-flow integrity) due to the presence of exploitable software bugs. Fortunately, our memory lockdown technique enforces a simple invariant by disallowing double mapping and any changes to the permission bits (e.g., R/W and NX) associated with the hypervisor code and data. A more subtle attack is that the attacker might map the hypervisor's memory to a compromised guest VM (another variant of

double mapping) by manipulating shadow page table entries. HyperSafe effectively blocks this by write-protecting the shadow page tables and preventing it from happening.

Also, instead of focusing on subverting page tables, the attacker may simply misuse existing code for malicious computations – as demonstrated by recent return-oriented programming [20], [43]. We point out the recently-surfaced return-oriented attacks are the very reason behind the expanded HyperSafe protection from the code integrity to the control-flow integrity. Specifically, with restricted pointer indexing, HyperSafe protects the control-data and ensures their uses will always adhere to the control flow graph that is pre-computed a priori. It may be argued that a *ret* may be manipulated to return to another return address that is contained in its target table but not in the last call site. However, such an attack is seriously limited in its scope and capability due to the need to follow the pre-determined control flow graph. Also, recent efforts (e.g., WIT [2]) can be naturally integrated for improved precision and protection coverage.

To the best of our knowledge, HyperSafe is the first system that is able to provide hypervisor control-flow integrity. This guarantee is achieved by creating an unbreakable deadlock for the attackers. Specifically, the deadlock is centered on the need to subvert the page table for the attackers: On the one hand, to manipulate the page table, the attackers need to execute a turn-off-WP instruction injected or misused out of the normal control flow. On the other hand, to execute the turn-off-WP instruction injected or misused out of the normal control flow, they need to hijack the execution and tamper with write-protected code or control-flow data, which in turn requires manipulating the page table. By integrating recent TPM-assisted static/dynamic root of trust [21], [48] that establishes load-time integrity, HyperSafe effectively enables non-subvertable enforcement for lifetime integrity.

B. Synthetic Experiments

To further validate the HyperSafe's design, we empirically evaluated its effectiveness against several powerful synthetic attacks. Specifically, we deliberately introduced a hypercall interface with various buffer overflow vulnerabilities and ported the Wilander's buffer overflow benchmark testsuite [54] for a number of realistic attack scenarios. By exploiting these vulnerabilities, the attacker can write to arbitrary memory locations with any value of choice. In our experiments, we have conducted four different types of attacks: the first one modifies the hypervisor code, the second one executes from the injected code, the third one modifies the page table, and the fourth one tampers with a return pointer. Note these attacks mimic the key techniques of the real world attacks against hypervisors as shown in the National Vulnerability Database [33]. Our experiments show that HyperSafe successfully prevented all of them.

More specifically, in the first experiment, we tried to overwrite one of the hypervisor's instructions with the instruction to reload the $CR0$ register so that the WP bit can be turned off. The write operation immediately triggered a page fault exception with the error code $0x03$. This error code indicates that the fault was caused by an illegal memory write and the register $CR2$ contains the address of the faulting memory write. In the second experiment, we spilled a sequence of code (that turns off the WP protection) to a global array in the heap and the exploit triggered the execution of the spilled code. This attack is successfully foiled by the HyperSafe's NX protection as the execution attempt leads to a page fault exception with an error code $0x11$. This error code reports that the page fault was caused by an NX violation. In the third experiment, we targeted the page table by attempting to make the previously mentioned array executable. We point out this attack challenges the HyperSafe's key technique – non-bypassable memory lockdown – and *will be successful in a hypervisor if not protected by HyperSafe*. Fortunately, with HyperSafe, the hypervisor's page tables are write-protected, and the page table update attempt triggered a page fault with the error code $0x03$. As compared to the first experiment, the faulting address (contained in $CR2$) in this case now pointed to one of the page table entries. Lastly, in our fourth experiment, we attempted to alter the hypervisor's control flow by modifying a return pointer on the stack. Interestingly, HyperSafe silently defeated and recovered from the attack. A further investigation showed that the attacked return pointer belongs to a function, which is called only from one location. Recall the optimization that avoids the unnecessary memory reads for performance (Section III-B), the instrumented code essentially ignores the return pointer on the stack and uses a direct *jmp* instruction to return to its (single) caller. When the optimization is not applied, the modified return pointer (or more precisely pointer index) will return back to either the original caller or $error$. In either case, this attack is foiled.

Also, we point out that once an attack is detected, we can easily combine the knowledge of the page fault's error code, the faulting address, and the hypervisor's memory layout to infer the nature of captured attacks. For example, if the faulting address $CR2$ points to an entry in a page table and the error code is $0x03$, we can tell that the attacker intended to manipulate the hypervisor's page tables. Based on the nature of the captured attack, we can then determine the most appropriate response. In our prototype, we simply issue an alert message, dump the machine context, and recover the execution if possible by ignoring the page faults.

C. Performance Evaluation

To evaluate the performance overhead introduced by HyperSafe, we measured the runtime overhead with standard benchmark programs including LMbench [31], UnixBench [51], ApacheBench [4], and two other real world applica-

Table I
SOFTWARE CONFIGURATIONS FOR EVALUATION

Item	Version	Configuration/command
Ubuntu Desktop	9.10-AMD64	standard installation
Clang/LLVM	2.6 pre-release2	default configuration
LMbench	3.0-a9	make results see
UnixBench	4.10	./Run
Kernel Build	2.6.31.4 (59MB)	make allnoconfig && make
bzip2	1.0.5	tar -jxf <*kernel file*>
Apache Server	2.2.12	Ubuntu package
ApacheBench	2.3	ab -c3 -t 60 <*url*>

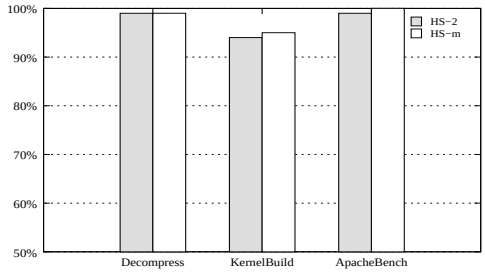


Figure 6. Normalized performance of application benchmarks with the original BitVisor as the baseline

tions. Our testing platform is a Dell Optiplex 755 desktop with a $3.0GHz$ Intel E8400 Core2Duo processor and $3GB$ memory. The machine runs a default installation of the Ubuntu 9.10 desktop with the official $2.6.31.14$ kernel. Table I shows the configurations of our evaluation platform.

We tested the guest OS's performance under BitVisor in two scenarios: *with* and *without* the HyperSafe protection. In order to further evaluate the impact from the improved precision, we tested two different implementations of HyperSafe: one has the least precise implementation with two big target tables (one for return instructions and one for indirect calls), and another has the most precise implementation with one target table for each function and one target table for each indirect call instruction. For simplicity, the former prototype is represented as *HyperSafe-2*, and the latter as *HyperSafe-m*. Note that the *HyperSafe-m* prototype includes the optimization mentioned in Section III-B.

In our evaluation with the application benchmarks, we calculated the running time with the Linux *time* command and reported the system time plus user time. In the ApacheBench test, we run the Apache server inside the Ubuntu 9.10 desktop and the ApacheBench client on another Dell machine with the same hardware configuration to measure the web server's throughput. These two machines were interconnected by a Gigabit Ethernet switch. All the test results reported here were the average of 10 runs. The deviations among these 10 runs are small ($< 3\%$).

Application Benchmarks: We first performed application-level tests to measure HyperSafe's impacts on real world programs. For that, we decompressed the official Linux 2.6.31.4 kernel source tarball, and then compiled the kernel. Conceptually, the kernel decompression is a computation-oriented task which will involve less hypervisor intervention, while the kernel compilation involves lots of device I/Os that will be intercepted by the hypervisor. As a result, we anticipate that the impact on the kernel compilation is more significant than the impact on the kernel decompression. The third test is the standard ApacheBench program that measures the throughput of an Apache web server running inside the Ubuntu system.

Table II
LMBENCH PERFORMANCE RESULTS (IN μs − SMALLER IS BETTER)

VMM	ctx	stat	mmap	sh proc	10K file	Bcopy
BitVisor	31.0	0.86	5379	5976	28.8	1377
HyperSafe-2	32.7	0.87	5411	6181	30.1	1451
overhead	5.5%	1.2%	0.6%	3.4%	4.5%	5.4%
HyperSafe-m	31.2	0.86	5249	5844	29.8	1416
overhead	0.6%	0%	-2.4%	-2.2%	3.5%	2.8%

Figure 6 shows the normalized performance results of HyperSafe-2 and HyperSafe-m when compared to the unmodified BitVisor. Overall, HyperSafe-m introduces less than 5% performance overhead. Interestingly, in all these tests, *HyperSafe-m outperforms HyperSafe-2!* This sounds counter-intuitive as HyperSafe-m achieves better precision than HyperSafe-2 likely at the cost of higher performance overhead. A further investigation indicates that the presence of multiple target tables and the optimization that avoids unnecessary memory reads when performing indirect control transfers both lead to improved performance. More specifically, the finer destination tables can improve cache utilization due to better locality. Also, the optimization in HyperSafe-m avoids the execution of additional memory-reading instructions.

Micro-benchmarks: We also used the standard micro-benchmark suites to evaluate HyperSafe's impacts on various aspects of OS operations. Here, we focus on the context switch and memory operation overheads as they are known to cause most performance impacts [46]. In Table II, we show the LMbench results. Specifically, the column *ctx* shows the latency of performing a context switch; the columns *stat* and *mmap* are the latency required to execute the corresponding system call; *sh proc* is the time spent to execute the C library function *system*; and *10K file* reports the time to create a 10KB file; and the column *Bcopy* shows the time to copy 1MB data using the C library function *bcopy*. For the UnixBench results, the final scores for BitVisor, HyperSafe-2, and HyperSafe-m, are 865.9, 844.5 (2.5%), and 849.6 (1.9%), respectively.

In general, BitVisor without the HyperSafe protection performs better than the two HyperSafe-based implementations. However, for two specific tests, i.e., the *mmap* and *sh proc* tests, HyperSafe-m actually performs better than BitVisor by a small margin, likely caused by the variations in our experiments. And consistent with the previous application-level tests, HyperSafe-m always beats HyperSafe-2 for the same reason as mentioned earlier: HyperSafe-m can achieve

better cache utilization and avoid unnecessary memory reads in certain control flow transfers. In other words, HyperSafe-m is not only more precise than HyperSafe-2, but also runs faster. As a result, the benefits in achieving an enhanced security guarantee and improving system performance are worth the extra development and debugging efforts to implement the more precise HyperSafe-m.

In conclusion, HyperSafe is a lightweight hypervisor protection mechanism that incurs less than 5% performance overhead, thus satisfying our third design goal (Section II).

V. Discussion

In this paper, we have so far discussed the protection of Type-I bare-metal hypervisors by proposing two key techniques. In the following, we examine the possibility of porting them to support other Type-II hosted hypervisors or commodity OS kernels. Note a type-II hypervisor requires a hosted OS kernel. Therefore, similar challenges will be encountered to enable their support. In the following, we discuss the challenges we may face in the process and examine possible limitations in the current prototype.

At first glance, Type-I hypervisors and commodity OSes (including the Type-II hypervisors) are both system software that directly run on top of the bare-metal hardware and can be similarly supported. However, certain choices made in commodity OS design and implementation present additional challenges to achieve the intended security guarantees. For instance, the design of modern OSes such as Linux put much emphasis on the performance. Developers use all kinds of hacks to squeeze more performance from the hardware. And unfortunately, not all of these techniques are sound in security. This is especially true in the area of memory management. Issues such as double mapping and mixed pages are quite common in commodity OS kernels (e.g., the Linux kernel always doubly-maps the lower memory). To implement the similar memory lockdown technique in HyperSafe, we need to remove all the doubly-mapped and mixed pages. In this process, the presence of doubly-mapped pages will likely cause more difficulties as it will require to overhaul the way the kernel accesses its memory. And conceivably, it is a challenging task to ensure the re-designed memory management can still achieve comparable performance while accommodating the much more frequent (benign) page table updates. Considering the need to escort page table updates and the associated overhead, the OS might consider implementing a batch mode for page table updates so that the cost can be amortized.

Additional challenges are also present to enforce the kernel's control-flow integrity, mainly due to the asynchronous nature of context switching and interrupt handling as well as the support of (potentially closed-source) third-party drivers. In particular, when a running process is being interrupted, the machine states are saved to memory so that they can be re-used later for resumption. Note these saved states can be potentially tampered with by the attacker to hijack the control flow. For that, there is a need to carefully examine all possible situations that may lead to states being saved to memory and ensure their correctness before being restored. Fortunately, most of these efforts are required only once. However, the support of loadable kernel modules, especially closed-source third-party drivers, remains a challenge. Specifically, for the enforcement of control-flow integrity, our second key technique requires a precise alias analysis. How to improve its precision when handling a large-scale system software such as commodity OS kernels as well as assembly instructions (or binary code) is still an ongoing research topic.

Also note that in its current form, our CFI enforcement is not as restrictive as possible because impossible paths [1] are still tolerated. For example, the indirect call site R1 in Figure 4 is allowed to transfer control to functions *func_i* and *func_j*. However, there may exist certain execution paths where only one of them is the valid target. Similarly, a return instruction may return to call sites other than its most recent caller. In Figure 4, the attacker may force function *func_i* to return to call site *R2* by manipulating the return index on the stack, even that the function should return to *R1*. To address impossible paths, one possible way is to make our CFI enforcement context sensitive. For instance, the shadow stack provides a viable way to enforce strict control transfer for returns. Unfortunately, our implementation experience (Section III-B) shows that performance overhead of write-protected shadow stack is high. However, despite these limitations, CFI still severely limits what attackers can achieve and is able to provide protection against a wide spectrum of attacks [1], [25].

For the very same reason, in our prototype, for some specific indirect function calls, we were forced to manually compute their call targets to handle the imprecision of the existing alias analysis tool we used. Manual analysis is tedious, time-consuming and error-prone. For the support of commodity OSes, there is a need to completely eliminate the need of manual involvement. To achieve that, we need to (1) scale the current field-and-context-sensitive unification-based alias analysis method and make it applicable for commodity OS kernels; and (2) enhance it for better precision by allowing for inclusion-based or flow-sensitive alias analysis and supporting assembly code. Note some promising progresses in this direction have been made by existing research efforts [6], [8], [11], [19] and the integration of these techniques remains an interesting direction for future work.

VI. Related Work

Program Analysis and Formal Proof The first area of related work is recent efforts in applying static analysis to identify and remove software bugs or using formal methods to prove certain security properties. For example, static

analysis, model checking, and symbolic execution have long been explored in the area of security research [2], [9], [10], [15], [17]. These systems are designed to uncover bugs in the source code [10], [15]; prevent the bug from being exploited [2]; reason about the safety of one facet of the software [9]; or demonstrate the absence of some kind of bugs [17]. Some of these systems can be scaled to analyze commodity OS kernels. However, they typically focus on a small subset of security vulnerabilities or properties. For example, Bugrara et al. [9] validates the safety of Linux kernel's pointer dereferences in system call arguments, where the safety is defined by the presence of sanity checks of pointers, not the proper use of the dereferenced contents.

There also exist parallel research efforts [3], [16], [26] that aim to formally prove the safety of system software. Among them, seL4 [26] recently made significant progress. In particular, it proves that the C code of the seL4 micro-kernel ($\sim$8700 SLOC) implements the behaviors specified in the abstract specification and contains nothing more. As mentioned earlier, the proof is achieved by imposing several restrictive requirements on the micro-kernel's design and implementation. Specifically, it requires interrupts being disabled for most of the time and instead schedules the interrupt polling at a small number of carefully-placed interrupt points. Further, it moves the memory management out of the kernel and avoids the need of verifying it as part of seL4. Due to these restrictions, it is still an open question of how well formal methods can be applied to commodity hypervisors such as Xen. (Note Xen 3.4.1 has $\sim$230K SLOC and its memory management alone has $>$14K SLOC). In comparison, HyperSafe takes a different approach to provide control-flow integrity even in the presence of exploitable memory corruption bugs. As such, both approaches are complementary in nature and our system can be leveraged to ensure the runtime integrity of a seL4 micro-kernel.

Protection of OS Kernels or Running Applications The second category of related work aims to protect the integrity of OS kernels or running applications. Systems such as [34], [35], [36], [40], [44], [53] take advantage of the isolation and dominant control provided by a trusted hypervisor to secure the integrity of guest OS kernels. With their assumption of a layer-below trusted entity to provide the base, the techniques in these systems cannot be directly applied for the hypervisor protection. In other words, HyperSafe can not enjoy the luxury of such an one-layer-below approach since it already runs at the lowest layer.

There also exist a number of other systems that have been developed to protect the control data from being hijacked. For example, StackGuard [13], StackShield [47], and others [29], [54] protect the return addresses from being hijacked or misused. Lares [34] and HookSafe [53] instead protect kernel hooks (or function pointers) inside the kernel space from being manipulated by rootkits. Note each system only partially meets the need in enforcing the entire control-flow integrity and their enforcement usually relies on a trusted entity. In contrast, our system enables the self-protection of hypervisors and provides non-bypassable enforcement of control-flow integrity.

The notion of control-flow integrity is initially proposed to protect user-level applications [1] by assuming two fundamental assumptions, i.e., non-writable code and non-executable data, from the underlying OS kernel. As a comparison, HyperSafe eliminates these two assumptions by proposing a non-bypassable memory lockdown technique. Without relying on external or layer-below components to provide the intended integrity guarantee, this technique serves as the foundation of our scheme and is thus one key contribution of this paper.

Hardware Support for Static and Dynamic Root of Trust Also closely related, the Trusted Computing Group [50] has provided foundational work such as Trusted Platform Module (TPM) [48] and Core Root of Trust for Measurement (CRTM) [50] that enabled trusted computing in commodity hardware. The recent Intel TXT technology [21] has provided a reliable way called measured late launch to securely load a clean hypervisor (or OS kernel). They have been leveraged to provide secure loading (with the guaranteed load-time integrity) [49], [32], integrity measurement [39], [22], and attestation [41], [42], [45]. HyperSafe relies on these works to ensure hypervisor load-time integrity and further complements them by effectively providing runtime integrity to the hypervisor.

VII. Conclusion

We have presented HyperSafe, a lightweight approach to provide lifetime control-flow integrity for commodity Type-I hypervisors. HyperSafe achieves its goal by two key techniques: The first technique locks down write-protected memory pages and prevents them from being manipulated at runtime, thus effectively protecting the hypervisor's code integrity; The second key technique converts the control data into pointer indexes by introducing one layer of indirection and thus expands protection to include control-flow enforcement. A proof-of-concept system has been developed to protect two open-source Type-I hypervisors: BitVisor and Xen. Experimental results with a number of (synthetic) hypervisor attacks as well as benchmarking programs show HyperSafe can reliably provide the intended security guarantee with a small performance overhead.

Acknowledgments The authors would like to thank the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. The authors are also grateful to Jinku Li, Michael Grace, Sina Bahram, Deepa Srinivasan, and Minh Q. Tran for useful discussions. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and the US National Science Foundation

(NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO and the NSF.

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, November 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, May 2008.

[3] E. Alkassar, M. Hillebrand, D. Leinenbach, N. Schirmer, A. Starostin, and A. Tsyban. Balancing the Load: Leveraging Semantics Stack for Systems Verification. *Journal of Automated Reasoning*, 2009.

[4] ApacheBench - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-To Analysis Using BDDs. In *Proceedings of the 2003 ACM SIGPLAN conference on Programming Language Design and Implementation*, June 2003.

[7] The Blue Pill Project. http://bluepillproject.org/.

[8] D. Brumley and J. Newsome. Alias Analysis for Assembly. Technical report, Carnegie Mellon University School of Computer Science, 2006. CMU-CS-06-180.

[9] S. Bugrara and A. Aiken. Verifying the Safety of User Pointer Dereferences. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, May 2008.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[11] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[12] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling Dynamic Program Analysis from Execution in Virtual Environments. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[13] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, , and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, August 1998.

[14] CVE-2008-0923. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0923.

[15] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[16] R. J. Feiertag and P. G. Neumann. The Foundations of a Provably Secure Operating System (PSOS). In *Proceedings of the National Computer Conference*, 1979.

[17] J. S. Foster, M. Fhndrich, , and A. Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN conference on Programming Language Design and Implementation*, May 1999.

[18] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed System Security Symposium*, Febuary 2003.

[19] B. Hardekopf and C. Lin. Semi-Sparse Flow-Sensitive Pointer Analysis. In *Proceedings of the 2009 ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, January 2009.

[20] R. Hund, T. Holz, and F. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.

[21] Intel Trusted Execution Technology. http://www.intel.com/technology/security/.

[22] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, June 2006.

[23] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection Through VMM-based "Out-Of-the-Box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[24] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. SubVirt: Implementing Malware with Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, May 2006.

[25] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[26] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 2009.

[27] L4.verified. http://ertos.nicta.com.au/research/l4.verified/.

[28] A. Lanzi, M. Sharif, and W. Lee. K-Tracer: A System for Extracting Kernel Malware Behavior. In *Proceedings of the 16th Network and Distributed System Security Symposium*, February 2009.

[29] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits with "Return-less" Kernels. In *Proceedings of the 5th ACM SIGOPS EuroSys Conference*, April 2010.

[30] The LLVM Compiler Infrastructure. http://llvm.org/.

[31] LMbench - Tools for Performance Analysis. http://www.bitmover.com/lmbench/lmbench.html.

[32] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the 3rd ACM SIGOPS EuroSys Conference*, April 2008.

[33] National Vulnerability Database. http://nvd.nist.gov/.

[34] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 29th IEEE Symposium on Security and Privacy*, May 2008.

[35] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[36] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th Recent Advances in Intrusion Detection*, September 2008.

[37] R. Riley, X. Jiang, and D. Xu. Multi-Aspect Profiling of Kernel Rootkit Behavior. In *Proceedings of the 4th ACM SIGOPS EuroSys Conference*, April 2009.

[38] T. Roscoe, K. Elphinstone, and G. Heiser. Hype and Virtue. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, May 2007.

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

[40] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st ACM ACM Symposium on Operating Systems Principles*, October 2007.

[41] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[42] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[43] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, October 2007.

[44] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[45] E. Shi, A. Perrig, and L. van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[46] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, March 2009.

[47] Stack Shield. http://www.angelfire.com/sk/stackshield/info.html.

[48] Trusted Computing Group: Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[49] Trusted Boot. http://tboot.sourceforge.net.

[50] Trusted Computing Group. http://www.trustedcomputinggroup.org/.

[51] UnixBench. http://ftp.tux.org/pub/benchmarks/System/unixbench.

[52] VMware ESXi: Bare Metal Hypervisor. http://www.vmware.com/products/esxi.

[53] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, November 2009.

[54] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, Febuary 2003.

[55] R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning. http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf.

[56] R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. In *Black Hat DC*, Febuary 2009.

[57] R. Wojtczuk, J. Rutkowska, and A. Tereshkin. Xen 0wning Trilogy. http://invisiblethingslab.com/itl/Resources.html.

[58] Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1, 2009.

[59] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the 16th Network and Distributed System Security Symposium*, Febuary 2008.