TF-BIV: Transparent and Fine-grained Binary Integrity Verification in the Cloud

Fangjie Jiang[1,2,3], Quanwei Cai[1,2*], Jingqiang Lin[1,2,3], Bo Luo[4], Le Guan[5], Ziqiang Ma[1,2,3]
1. State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
2. Data Assurance and Communication Security Center, Chinese Academy of Sciences
3. School of Cyber Security, University of Chinese Academy of Sciences
4. Department of Electrical Engineering and Computer Science, the University of Kansas, USA
5. Department of Computer Science, the University of Georgia, USA
{jiangfangjie, caiquanwei, linjingqiang}@iie.ac.cn, bluo@ku.edu, leguan@cs.uga.edu, maziqiang@iie.ac.cn

ABSTRACT

With the emergence of virtualization technologies, various services have been migrated to the cloud. Beyond the tenants' own security controls implemented in the virtual machine (VM), the binary integrity verification mechanism in the virtual machine manager (VMM) provides stronger protections against malware. Unfortunately, none of existing integrity verification mechanisms in the cloud provides complete transparency and fine-grained efficiency. Some schemes selectively check the integrity of sensitive binaries, but they require modifications to the VMs (e.g., integrating monitoring libraries) to trigger verification. Others, although need no modification to the VMs, have to enforce checking on all the binaries, because they cannot distinguish binary images for the sensitive processes from the binaries for insensitive ones, leading to significant performance overheads. In this paper, we present TF-BIV, a transparent and fine-grained binary integrity verification scheme, which does not require any modification or software/driver installation in the VM. TF-BIV identifies the sensitive processes at the creation, and checks the integrity of the binaries (including the guest OS kernel and the dependant binaries) related to these processes. The provided transparency and efficiency are achieved by leveraging existing hardware virtualization supports (i.e., Intel extended page table) and debugging features (i.e., monitor trap flag). We have implemented the TF-BIV prototype based on QEMU-KVM. To demonstrate the usability of TF-BIV, we adopted it for cloud-based cryptographic services, to achieve the strict invoking controls. In addition to the password-based authentication, TF-BIV further achieves process-level authorization to the invokers. Intensive evaluation shows that TF-BIV implements the designed binary integrity verification with only about 3.6% performance overhead.

*Quanwei Cai is the corresponding author.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359795

CCS CONCEPTS

• **Security and privacy → Access control**; **Virtualization and security**.

KEYWORDS

Integrity verification, virtualization, cloud-based cryptographic service

ACM Reference Format:
Fangjie Jiang, Quanwei Cai, Jingqiang Lin, Bo Luo, Le Guan and Ziqiang Ma. 2019. TF-BIV: Transparent and Fine-grained Binary Integrity Verification in the Cloud. In *2019 Annual Computer Security Applications Conference (ACSAC'19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3359789.3359795

1 INTRODUCTION

In the past decade, cloud computing has been becoming the *de facto* standard model of hosting and delivering online services. While migrating services to the cloud brings significant benefits such as scalability and cost reduction, security has been a major concern. All stakeholders in this scenario have their own security considerations and, in response, deploy their own controls. First, it is critical for the tenants to ensure the integrity of their core services hosted in the cloud. Conventional security controls, such as anti-virus, web application firewall, and OS hardening are deployed within the virtual machines (VMs). However, they might be unable to deal with zero-day vulnerabilities [23–25], misconfigurations, phished users, etc. On the other hand, cloud service providers (CSPs) are concerned with the security of the tenants' systems, as well as the security of the cloud platform itself. In particular, a compromised guest process/OS may escape from the VM to harm the host OS [84, 85], to perform malicious tasks such as Bitcoin mining [53], or launch attacks against other tenants [41, 62]. Meanwhile, the traditional access control of sensitive services (e.g., cryptographic service) in the cloud is based on the identifier of the VM and the corresponding password, but fails to achieve the control based on the invoking process, while only a small number of processes are actually authorized to invoke the sensitive service. Therefore, the malicious processes running in the victim VM may invoke the sensitive service with leaked passwords.

A mechanism that consistently validates the integrity of the sensitive VM binaries (including OS kernel and all dependant libraries) will benefit both the cloud tenants and service providers. The CSPs could adopt this mechanism to monitor the status of processes running in VM, and to effectively identify malicious or compromised processes. Moreover, the cloud platform could provide the integrity verification as an add-on service to the tenants, i.e., binary-integrity-as-a-service. The tenants will benefit from another layer of protection beyond their own security controls. Since the CSP-provided mechanism runs within the VMM, the integrity monitor will still function even when the guest OS is compromised and/or security functions in the guest OS fail.

Various mechanisms have been proposed to check the integrity of the OS and applications running in a guest VM. Patagonix [54] and HIMA [5] are proposed for the sensitive scenario, in which all active processes in the VM are validated. To guarantee the integrity of specified processes, InkTag [36] has to modify the guest OS kernel and AppShield [19] integrates a secure module into the sensitive applications, which constructs an isolated context for high-assurance processes even in the presence of compromised OS. En-ACCI [47] validates the integrity of the invoking processes to cryptographic services in the cloud. However, it only validates the invoking process when the cryptographic service is being invoked but not beyond, making it vulnerable to Time of Check to Time of Use (TOCTTOU) consistency attacks. In summary, no binary integrity verification scheme has yet been developed that meets the following four desired properties.

- **Isolation**: The integrity verification system is fully isolated from the guest OS and the target applications. The in-VMM validator should not be interfered by the malicious guest OS or target applications.
- **Transparency**: The guest OS and the target application should be oblivious of the security checking conducted in the VMM. Thus, no modification (e.g., hooks, drivers, or libraries) is required in the guest OS.
- **TOCTTOU consistency**: The continuity of integrity guarantee beyond the time-of-verification must be provided.
- **Fine-grained verification**: Given that there are more non-sensitive applications than sensitive ones in a VM, it is desirable that the tenants have the flexibility to designate the sensitive applications to be protected. So the tenant is free to update the non-sensitive binaries in bulk independently. For sensitive binaries, the tenants have to notify the CSP of any changes to sensitive binaries so that an up-to-date integrity meta-data of the sensitive binaries are always available in VMM.

In this paper, we present TF-BIV, a binary integrity verification scheme in the cloud. As far as we know, TF-BIV is the first to achieve the aforementioned properties. More specifically, (1) to achieve isolation, in TF-BIV, integrity verification is triggered by hardware events and is performed in VMM. Therefore, subverted applications or OS kernel in the guest VM cannot interfere with or bypass TF-BIV. (2) To keep the guest VM unmodified, TF-BIV leverages hardware virtualization and virtual machine introspection (VMI) to hook critical system events (e.g., process creation) and to infer VM semantic (e.g., the virtual memory layout), respectively. Note that the result of VMI might be untrusted. We *never* use it to influence the verification results, but only for acceleration purposes. (3) TF-BIV leverages hardware-assisted second-level address translation (e.g., Intel EPT) and monitor trap flag (MTF) [42] to continuously monitor any updates to the page table and the verified physical pages of the target process. As a result, any modification to the verified memory pages (and the memory layout) will be captured by hardware. This ensures TOCTTOU consistency. (4) Finally, all of these security checks are performed on a per-process basis. Unprotected processes are not influenced. Thus, fine-grained process-level verification is provided. Meanwhile, the basic shared system services (i.e., the OS kernel, the runtime and drivers) are verified by default.

The process-level integrity monitoring is especially important for critical or sensitive add-on services provided by cloud platforms. For tenants, the invoking process requires almost the same security assurance as the sensitive service (e.g., cloud-based cryptograhic services). CSPs are also concerned with the security of the invoking process. Once the invoking process is compromised, the add-on services may be abused. For example, the RSA private key may be used to sign a fake X.509 certificate. Typical cryptographic services in the cloud include (1) the key management service (KMS), such as AWS KMS [2], Azure Key Vault [55], Google Cloud KMS [34] and Aliyun KMS [20]; and (2) the cloud-based hardware security module (Cloud HSM), such as AWS Cloud HSM [1] and Google Cloud HSM [33]. CSPs strive to ensure the security of the cryptographic services, for example, they adopt FIPS 140-2 [30] accredited hardware security modules for the cryptographic computations. However, the security of the invoking process (i.e., the invoker) is often overlooked: (1) the identity of invokers and the corresponding passwords used in authentication may be leaked due to various reasons [13, 75]; (2) only one or a small number of processes running in the VM actually need to invoke the cryptographic service, but access to the service is authorized to the entire VM; and (3) an adversary may gain access to the victim VM (on the whitelist), or tamper with a vulnerable process to invoke sensitive services, bypassing authentication and authorization. Therefore, to ensure the secure invoking of sensitive cloud services, the CSPs are expected to verify the integrity of the caller process, including the OS kernel and all dependant libraries. As a side benefit, authorization of callers of sensitive services will be specified and enforced at the process level.

The main contributions of TF-BIV are as follows:

- We propose a hardware-assisted binary integrity verification scheme in the cloud. It supports fine-grained protections that allow tenants to specify the processes to be validated. The proposed mechanism is completely transparent to the VM, and it does not need any support (OS modification, configuration, driver/software installation, or modules integration) from the guest OS and applications.
- We developed a prototype of TF-BIV on QEMU-KVM with Intel EPT. With extensive experiments, we demonstrate the effectiveness and efficiency of TF-BIV. We also provide an in-depth security analysis of TF-BIV, and qualitatively compare TF-BIV with other binary integrity verification schemes.
- We apply TF-BIV as to check the integrity of processes that invoke cloud-based cryptographic service. The result shows that the overhead is modest. This demonstrates that TF-BIV effectively protects sensitive services by only allowing the access from authorized processes.

The rest of this paper is organized as follows. We introduce the background and the threat model in Sections 2 and 3. Section 4 describes the technical details of the integrity verification mechanism. We then present the security analysis in Section 5. We introduce the process-level authorization mechanism for cloud-based cryptographic services, as the application of TF-BIV in Section 6. Next, we present the implementation specifics and experimental results in Sections 7 and 8, respectively. Finally, we discuss the related works in Section 9, and then conclude the paper.

2 BACKGROUND AND PRELIMINARIES

TF-BIV relies on VMI and hardware-assisted virtualization to achieve transparent process-level binary integrity checking. This section provides necessary background information about VMI and Intel's hardware virtualization implementation – VT-x.

2.1 Virtualization and VMI

Virtualization allows multiple OSes to share a single physical machine. It creates an illusion that every OS runs in its own (virtual) machine. To ensure security, a virtual machine manager (VMM) is responsible for isolating VMs and controls access to hardware resources, so that VMs do not interfere with each other. Exceptions, e.g., I/O access and system events, are handled and mediated by the VMM.

In a typical virtualization environment, the VMM is oblivious of the computing tasks being conducted in the VMs. To increase the visibility to VMs, virtual machine introspection (VMI) [31] is proposed. Since VMI is conducted independently of the VM, it is ideal for security purposes, such as software patching without interrupting the execution of guest OS [9, 15], intrusion detection [31], forensics [29]. VMI takes advantage of the OS-specific knowledge to effectively inspect the internal states of each VM. In other words, it assumes certain internal data structures used by guest OS [31]. Unfortunately, this assumption of VMI could be exploited by malicious VMs [7]. For example, as demonstrated in DKSM [7], attackers could easily manipulate kernel data structures to mislead a VMI tool.

2.2 Intel Hardware-assisted Virtualization

Before hardware-assisted virtualization, virtualization has to be implemented by fully emulating an entire computer, leading to significant performance overheads. Modern processors integrate hardware support for accelerated virtualization (e.g., Intel VT-x). VT-x capability is identified by a `vmx` CPU flag, which stands for Virtual Machine Extensions. It adds new instructions and architectural support for a virtual execution mode where the guest OS perceives itself as running with full privileges and runs native instructions directly on the host without emulation.

The most important data structure in hardware virtualization is *virtual machine control structure (VMCS)*, which is maintained by hardware. It stores critical system events of interest for the guest OS. Whenever such an event happens, the VM is suspended and the execution returns to VMM. Providing such an interface that intercepts them gives VMM an opportunity to scrutinize the VM validity. Note that the corresponding fields in VMCS, e.g., *Monitor trap flag (MTF)* and *CR3-load exiting flag*, cannot be modified directly with load/store instructions. Instead, dedicated virtualization instructions have been incorporated that affect VMCS indirectly. For example, MTF [42] is a debugging feature. When set, the guest will trigger a VM exit after executing each instruction. As indicted by the name, when the CR3-load exiting flag is set, any operation causing an update to `CR3` register causes a VM exit.

Without hardware virtualization, VMM needs to maintain a shadow page table that maps the guest virtual addresses (GVA) to host physical addresses (HPA). Whenever the guest OS updates its page table, VMM needs to synchronize it to the shadow page table. With Intel hardware virtualization, extended page table (EPT) acts as a second layer of address translation. EPT translates GPA into HPA in hardware directly without the intervention of VMM. This not only simplifies the VMM design, but also improves performance. Like the traditional page table, the EPT paging-structure entries contain the privilege flags (i.e., read, write and execute) for the corresponding GPA. If a memory access violates the specified privileges, the VM exits with the reason code "EPT violation". To further increase performance, the VMX extension supports virtual-processor identifier (VPID). With VPID, the logical processor includes a tag in the translation lookaside buffer (TLB) that identifies the corresponding process. As a result, the hypervisor does not need to flush TLBs during context switches.

3 THREAT MODEL

Assumptions. In TF-BIV, as with many software-based protection mechanisms, we assume that all the hardware and firmware underneath the guest VM are trustworthy. More specifically, the physical processor provides basic hardware-based controls (e.g., memory management unit, task switching, privilege transition, etc.). Since we rely on hardware virtualization, the VMX extension is also assumed to work as expected. For instance, `VM entry` and `VM exit` do not leak any sensitive information to less privileged software stacks. VMCS can never be accessed by software, including VMM. MTF and EPT work as documented in the Intel manuals [42].

We assume that supervisor, together with VMX-enable processor, correctly provides an isolated environment for each VM. VMM manages system resources and mediates access to these resources as programmed. The protection of VMM is out of the scope in this work. However, various solutions (e.g., HyperSafe [82], HyperCheck [81] and XMHF [78]) are ready to be deployed.

TF-BIV focuses on the integrity protection of static code segments. Verification of dynamically generated code, such as that generated with Just In Time (JIT) engines and Dynamic Binary Translation (DBT), is out of the scope of this paper. Moreover, the bugs in the protected programs themselves (e.g., logic errors, memory disclosures) are not considered, since they do not modify the executable codes. These bugs can lead to both control-oriented (e.g., control flow hijacking) and data-oriented attacks [17, 37, 39, 44]. In both cases, the behavior of the program is changed. To defeat against such software bugs, there are many orthogonal mechanisms including control flow integrity [49, 74], data flow integrity [12, 72], and whole memory safety [27, 57].

TF-BIV relies on the cryptographic hash to verify the integrity of binaries. We assume that the adopted hash functions are secure and immune to the collision attacks. Moreover, we assume that the reference hash values of the binaries are calculated in a secure environment and imported to the CSP securely with out-of-band channels.

Threat Model. The attacker's ultimate goal is to execute a malicious copy of the sensitive process (called S-process in this paper). He could target any process in the distribution, deployment and execution of the program. For example, he could intercept the network and modify the binary of the program during software downloading. After the software has been deployed, he could also inject code into the address space of the S-process at run-time. Even in the correctly bloated Linux kernel, there are multiple approaches to achieve this. For instance, he could exploit vulnerabilities in the OS kernel to control the kernel [83]. Then he could inject new code or manipulate existing code of the S-process during execution.

As mentioned above, we have integrated TF-BIV into a cloud-based cryptographic service. In this scenario, the attacker's goal is to obtain access to cloud-based cryptographic service even he is not authorized. He may obtain the identity and credential information to access cryptographic computations using social engineering. He also has the capability to bypass the existing access control of the cryptographic service and invoke requests to cryptographic service within the VM. This capability is consistent with the aforementioned assumption that the attacker can take over of the entire guest OS.

4 TF-BIV DESIGN

In this section, we depict the design of TF-BIV.

4.1 Design Goals

We design TF-BIV with four goals in mind, corresponding to the four desired properties of a integrity verification scheme mentioned in Section 1. Specifically, TF-BIV protects sensitive processes in a separated execution domain (**Goal 1: Isolation**) and does not need any modification to the guest VM (**Goal 2: Transparency**). Moreover, the integrity of a sensitive process is guaranteed throughout the entire lifetime of an S-process (**Goal 3: TOCTTOU consistency**). Finally, a tenant has the flexibility to protect only a particular set of programs (**Goal 4: Fine-grained verification**).

4.2 Overview

TF-BIV relies on pre-computed hash values for target binaries as a reference for integrity checking. Therefore, these hash values are securely computed in a separated machine and transmitted to the VMM via out-of-band channels. To calculate the reference hash values, TF-BIV needs to analyze the target binary and obtain all the dependant shared libraries, because all of them contain executable code in the address space of the target program. When the binary needs to be updated, the corresponding reference hash values should also be re-calculated and updated.

At run-time, TF-BIV, running inside the VMM, verifies the integrity of the specified process (S-process) according to the reference hash values. To achieve the design goals mentioned in Section 4.1, TF-BIV transparently detects four types of critical system events occurred in the guest VM, as shown in Figure 1. ❶ TF-BIV monitors the creation of a process by capturing changes in the CR3

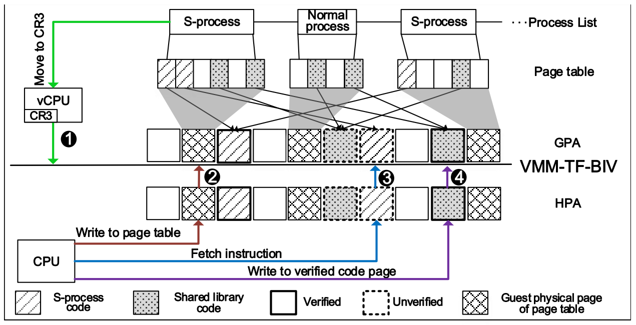

Figure 1: TF-BIV architecture.

register. This can be used to identify an S-process creation when it is activated for the first time. ❷ TF-BIV monitors all updates to the page table of an S-process. In this way, we track the code pages (including S-process and the dependant libraries) mapped into the virtual address space of an S-process. ❸ TF-BIV captures any execution attempts of S-processes. Before the code can be executed, TF-BIV must verify its integrity. ❹ TF-BIV captures any modification to the verified code pages. In this way, we ensure TOCTTOU consistency.

All of these four events will be detected transparently in VMM. This is achieved by configuring the corresponding flags in VMCS and EPT, which trigger a VM exit whenever one of the corresponding system events happens. In particular, TF-BIV registers the following four events. **a) CR3-load exiting**: TF-BIV tracks the S-process creation by monitoring any update to the CR3 register [40]. **b) MTF VM exiting**: TF-BIV monitors any update to the page table of S-processes. To achieve this, TF-BIV sets the first-level page table as non-writable. Any update to it triggers a VM exit. To be able to continuously track the whole page table data structure, TF-BIV leverages MTF to execute the relevant guest instructions in single-step mode. After the operation is finished, the whole page tables (including the initial first-level page table and upper level page tables discovered later) are set as non-writable again so as to monitor any further modifications. **c) EPT non-executable (NX) exiting**: TF-BIV employs this event to trigger the actual code integrity verification. In details, TF-BIV keeps track of all the unverified code pages of S-processes and sets the non-executable bit (NX-bit) of the corresponding EPT entries, making the execution of any unverified code trigger a VM exit. In VMM, TF-BIV verifies the corresponding code pages before execution. **d) EPT non-writable (NW) VM exiting**: For the verified code pages, TF-BIV sets the non-writable bit (NW-bit) of the corresponding EPT entry to ensure the TOCTTOU consistency. Any modification to a verified page triggers a VM exit. If this happens, TF-BIV marks this page as unverified so that later execution attempts would trigger another verification.

TF-BIV verifies the integrity of all the codes related with the S-processes, including the kernel, loadable kernel modules (LKM) and the shared libraries. For kernel integrity, TF-BIV follows the same approach as Patagonix [54]. It verifies kernel integrity before execution by comparing the hash values of kernel code with the reference hash values. Please see Patagonix [54] for more details about this static image. In the next, we detail how TF-BIV captures the aforementioned critical system events and transparently verifies the integrity of S-processes. We also discuss how to perform integrity verification for LKMs.

4.3 S-processes Identification

In the Linux OS, to execute a process, the kernel needs to set the CR3 register to point to the physical address of the page table of the target process. Therefore, monitoring updates to CR3 allows TF-BIV to capture the schedule of an S-process. TF-BIV will be transparently notified of a CR3 update, thanks to hardware virtualization support, in particular, the `CR3-load exiting` bit in `VMCS`.

TF-BIV needs to further identify if a newly scheduled process is an S-process or not. For each scheduled (activated) process, TF-BIV firstly checks whether the process is a newly created one by comparing the new CR3 value with the list of all the previously recorded CR3 values. If a new process is identified, we could simply obtain the name of the process by examining the `comm` data structure. However, as kernel data structures might be manipulated, we directly use code hash values to match an S-process. Specifically, TF-BIV obtains the physical address of the code pages by traversing the page table, and compares the hash values of the code pages with the reference hash values to find the matching binary. TF-BIV keeps a record of all the S-processes into S-process list by their CR3 values.

4.4 Memory Layout Monitoring

To verify the integrity of the S-process thoroughly, TF-BIV needs to identify all the code pages mapped into the S-process's virtual address space. In details, TF-BIV firstly observes the newly mapped code pages by monitoring any modification of the S-process's page table, and then finds which binary the code page belongs to, and finally checks its integrity based on the corresponding reference hash values.

Monitoring page table updates. TF-BIV leverages the MTF exiting and EPT non-writable (NW) exiting to achieve non-bypassable monitoring of the update to an S-process's page table. Firstly, TF-BIV finds the memory pages for the S-process based on the CR3 value and sets S-process's page table as non-writable in EPT. As a result, when the guest OS attempts to update an S-process's page table, a VM exit occurs. In VMM, TF-BIV modifies the corresponding EPT entries for that update to allow temporary modification to the page table. It also sets the MTF flag in the `VMCS` structure to continuously monitor the following updates to the page table. After all the updates complete, TF-BIV clears the MTF flag and sets the page table as non-writable to monitor further modifications.

Identifying newly mapped memory areas. If a new code page is observed to be mapped in the address space, we have to find out which binary contains this code page to obtain the reference hash values for comparison. As the Address Space Layout Randomization (ASLR) has been widely adopted in modern OSes, we cannot use virtual address to locate a code section within the binary image of a protected program. A straightforward solution is to compare the hash values of these code pages with the reference hash values of all the code pages for each binary related with the S-process to identify the mapping between memory areas (i.e., the start and end GVAs) and the corresponding binary files. However, this is time-consuming.

TF-BIV solves this problem by extracting necessary semantic information of the guest OS with VMI. In the Linux kernel, the mapping between the virtual address of the memory area and the binary is recorded in the `mm.mmap` data structure. TF-BIV constructs a mapping between the memory areas and the related binaries based on this information. TF-BIV also constructs a mapping between the GPAs and the memory areas, and establishes the mapping between the virtual memory page and the code pages of the related binaries based on the offsets between the GVA and start GVA. Therefore, for a newly mapped memory area, TF-BIV can find the corresponding references hash values based on these mappings efficiently. Note that the adversaries can manipulate the kernel data structure to fool TF-BIV. However, we *never* use it to influence the verification results, but only for acceleration purposes. It only leads to DoS attack because it fails the integrity checking of the S-process as detailed in next section. Unverified code can never be executed in the S-process's address space.

4.5 Integrity Verification of Code Pages

TF-BIV configures the EPT entries of the code pages related to the S-process to enforce the W⊕X property, so that a code page can be executed only if it has been verified. In details, TF-BIV sets the EPT entry for each newly loaded physical page as non-executable, which causes a VM exit once an instruction fetching is requested on this page. TF-BIV then checks the integrity of this code page. If the verification passes, it configures the code page as executable but non-writable. Any modification to the verified code page will trigger a VM exit, so that the aforementioned verification is performed again.

TF-BIV differentiates the handling of EPT NX exiting depending on which code page is fetched. The code page is classified into three categories: kernel code, S-process related code (the code being mapped in the S-process's address space), and insensitive code (all others). TF-BIV considers CR3 value, GVA and GPA in classifying the code pages. As shown in Figure 2, TF-BIV uses GVA to determine whether the code page is the kernel code. If the code page is in user space, TF-BIV checks whether the code page belongs to any S-processes based on the CR3 value. For the code pages of an S-process, TF-BIV further distinguishes the code of the application itself or that of a shared library based on the mapping between memory areas and programs' memory mapped binaries (Section 4.4). If the CR3 register is not in the S-process list, TF-BIV uses the mapping between programs' memory mapped binaries and the guest physical pages mapped in the S-process's address space to determine whether the code page is related to any S-process. If not, this page is considered insensitive.

As shown in Figure 2, different types of integrity verification are performed in the handler of the EPT NX exit event for various code type. For the kernel code, TF-BIV distinguishes the LKM from other kernel code. We discuss LKM verification in Section 4.6 and static kernel code verification in Section 4.2. For S-process related code, TF-BIV obtains the information of the corresponding binaries and the offsets as discussed in Section 4.4, and checks the integrity

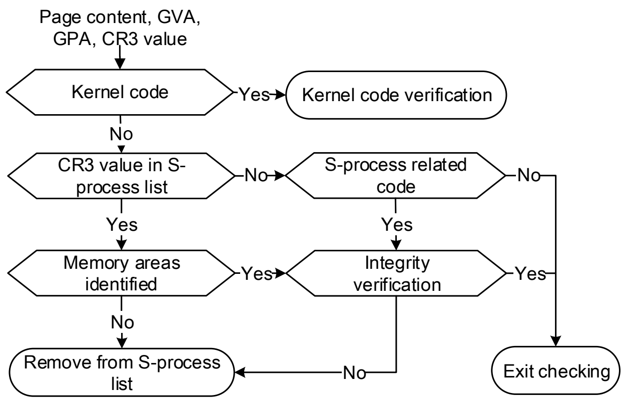

Figure 2: The handler of EPT NX exiting.

by comparing the hash values of the code pages with the reference hash values.

Before the S-process related code pages are mapped into S-process's address space, it may have already been executed (e.g., the same physical page is mapped to another process). To ensure no unverified code will be executed in S-process's address space, TF-BIV records the information of the newly mapped code in GPA list, and configures the corresponding EPT entry to make it non-executable again. Each entry of the GPA list is in the form of (`id`, `offset`, `GPA`, `state`, `num`), where `id`, `offset`, `GPA`, `state` and `num` denote the identifier of the binary which the code belongs to, the offset of the code page in the binary image, the GPA of the code page, whether the code page has been verified and the number of S-processes that this code page is mapped to, respectively. When an S-process related code is mapped into the address space of the S-process, if no corresponding entry exists in the GPA list of S-process related code, TF-BIV creates a new entry and sets `num` to one. Otherwise, it increases `num` by one.

In the handler of EPT NX exit events, TF-BIV checks `state` of the entry to check whether the code page has been verified or not. It avoids the redundant verification when the code is shared by multiple S-processes. When a PTE for an S-process is flushed, `num` is decreased by one. When `num` becomes zero, the whole entry is deleted.

4.6 Integrity Verification of LKM

An LKM is a dynamically loaded kernel component. When loaded, it is located in the free area of the kernel space. Code in an LKM is either loaded in the `init` block or `core` block. Pages belonging to the `init` block are executed only once during the initialization of LKM. They are erased after execution. Pages belonging to the `core` block are used throughout the lifetime of the LKM. They are not erased until the LKM is unloaded.

To identify which LKM the target code page belongs to, TF-BIV maintains a list in the form of ($init_s$, $init_e$, $core_s$, $core_e$) for all LKMs. In each entry of the list, $init_s$ and $init_e$ indicate the start and end GVA of the `init` block, and $core_s$ and $core_e$ are for the `code` block. When a code page is loaded, TF-BIV checks whether the GVA of this page belongs to any entry in the maintained list. When a match is identified, TF-BIV validates the calculated hash value against the corresponding reference hash values. If no match is found, TF-BIV checks whether the page belongs to a newly loaded LKM, comparing the calculated hash value with the reference hash values of the code pages which contains the entry point (initialization function) for each specified LKM, or the `core` block of an initialized LKM, and comparing the potential reference hash values based on the GVA of this code page and each $init_s$ in the list. If any match exists, TF-BIV updates the corresponding ($init_s$, $init_e$) or ($core_s$, $core_e$) entry. For an LKM that does not have code segment in its `init` block (e.g., `garp.ko` and `stp.ko`), TF-BIV compares the code page with all the code pages of each specified LKM, and inserts into the list a new entry (($core_s$, $core_e$)) if a match found. Otherwise, the validation fails, and the code page can never be executed. An entry is removed from the list when any page in the memory area ($core_s$, $core_e$) is modified, because it indicates that the corresponding LKM is unloaded.

4.7 Handling Mixed Pages

Mixed page problem is common in legacy OS kernel and applications. Although, various mixed page eliminations (e.g., page-aligning data) are proposed and even deployed in Linux kernel, mixed kernel page still exists [61]. In essence, a mixed page contains both code and mutable data, which conflicts with the basic assumption of TF-BIV.

When a mixed page is loaded, TF-BIV makes a duplication of its original physical page, and replaces all non-code areas with `NOP` instructions in the duplicated page. Verification is performed on the original page. If the verification is passed, TF-BIV configures the EPT entry for the original physical page as non-executable, but readable and writable, and sets the duplicated physical page as executable but non-readable and non-writable. At the very beginning, the mixed page is mapped to the original physical page, hence, executing any instruction from the mixed page triggers a VM exit. In the handler of this VM exit, TF-BIV re-maps the mixed page to the duplicated physical page to allow execution. Then, accessing data in the mixed page will trigger another VM exit, since the duplicated physical page is non-readable and non-writable. In the VM exit handler, TF-BIV maps the mixed page back to the original physical page.

Mixed page has been a problem with many similar solutions. For example, Patagonix [54] handles the problem with shadow page table. However, it incurs significant overhead due to the complexity of synchronization the shadow page table for maintaining the permission for the corresponding memory. TF-BIV is more efficient, as the permission is set on the GPA directly. Moreover, our protection does not result in frequent VM exits. This is because the mappings between the GPA to HPA for the instruction and data are cached in instruction TLB (iTLB) and data TLB (dTLB) separately, which avoids the VM exit when the corresponding iTLB and dTLB are not flushed.

5 SECURITY ANALYSIS AND COMPARISON

In this section, we analyze the security of TF-BIV, and compare it with other solutions.

5.1 Security Analysis

TF-BIV verifies all the code related with the S-processes, including the kernel code, LKM, the application code and shared libraries. The kernel code is verified during the start-up of the OS. The LKM is identified upon the loading and verified before execution. The code related to S-processes is verified before executing the code.

TF-BIV finds all these code pages based on the GVAs and in-memory page table. TF-BIV attempts to find the reference hash values for the code pages based on the kernel data structure (e.g., mmap) to accelerate the integrity verification. The adversary may modify the kernel data structure, which breaks the mapping between the code page and the reference hash values. However, in this case, the verification fails and the code page will never be executed. This only introduces the DoS attack but still ensures the goal that only the verified code will be executed in S-processes. The DoS attack may be mitigated by comparing the code page with all the reference hash values when it is inconsistent with the inferred reference hash values.

In the life cycle of an S-process, TF-BIV monitors and verifies each modification to all the related code. TF-BIV prevents the modification that occurs before the binary is loaded into memory, as the corresponding pages will yield an invalid hash values. At runtime, various attacks may be adopted to modify the integrity of S-processes or add new code pages. For example, the attackers can exploit vulnerabilities (e.g., buffer overflow and format string overflow [51]) of the S-process and others attacks (e.g., double mapping attack [19] and [63]) to inject malicious instructions and bypass the software protections (e.g., Stackguard [21], Stackshield [79], Formatguard [22], PaX [76], RSX [73] and kNoX [60]). TF-BIV prevents these attacks as it relies on EPT, the hardware features, to find the modification of code page in the address space of an S-process. Once any modification is detected, re-verification is enforced.

The adversary may also attempt to execute malicious code in S-process without modifying any code pages. For example, the address mapping manipulation attacks [19] may be adopted to map an unverified physical page to the S-process's address space, or map a physical page of an S-process to an unprotected process which makes the physical page executable without being verified. TF-BIV defeats these attacks by monitoring the page table of an S-process. Once a new physical page is mapped, TF-BIV finds this mapping operation immediately as the modification to S-process's page table triggers a VM exit. In processing this VM exit, TF-BIV sets the physical page as non-executable and classifies it as S-process related code, which ensures that the physical page will be verified before execution no matter which process it belongs to. More details are provided in Section 4.5.

Limitations. TF-BIV fails to provide protection for programs generated by the JIT engines and DBT, because they require the code pages to be both writable and executable. However, the interpreters themselves may still benefit from TF-BIV. TF-BIV fails to present software bugs in the programs themselves (e.g., CFI attacks, data-oriented attacks), because code integrity is not violated in these attacks. Existing mitigation mechanisms, such as Heisenbyte [74], PITTYAPT [28], KVM-PT [67], μCFI [38], CPI [49], YARRA [66], HardScope [58], HDFI [72] and DFI [72], may be employed for this purpose. The DMA attacks may modify the physical memory directly, bypassing the permission checking in MMU. As a result, code may be modified without causing a VM exit. IOMMU [69] may be integrated to prevent illegal DMA attacks.

Table 1: Analysis of existing integrity verification systems.

	Isolation	Consistency	Transparency	Fine-grained
Patagonix [54]	●	●	●	○
HIMA [5]	◐	●	◐	○
En-ACCI [47]	◐	○	●	◐
InkTag [36]	●	●	○	●
AppShield [19]	●	●	○	●
AppSec [59]	●	●	○	●
TF-BIV	●	●	●	●

○: not supported ◐: partially supported ●: fully supported

5.2 Comparison with Existing Solutions

Various hypervisor-based solutions [5, 19, 36, 47, 54, 59] have been proposed for the integrity verification and protection for different purposes. Patagonix [54] verifies all the binaries before execution to detect the rootkits. HIMA [5] prevents the execution of unauthorized binaries. InkTag [36], AppShield [19] and AppSec [59] protect the integrity of the critical process even when the guest OS is untrusted. En-ACCI [47] aims to enhance the authentication mechanism of the cloud-based cryptographic services by checking the integrity of the caller. Table 1 compares these solutions based on the design goals described in Section 4.1.

Isolation: The integrity validator needs to be isolated from the guest VM, and any activity in the guest VM should never harm the correctness of verification results. All these solutions aim to achieve this based on virtualization. Patagonix and TF-BIV achieve the full isolation, as Patagonix utilizes the VMM's control of the MMU while TF-BIV relies on the hardware feature to discover the code execution and trigger the integrity verification executed in VMM. However, HIMA requires kernel patch (PaX [76]) to enforce W$\oplus$X support, which may impact the verification result if the patching is subverted. En-ACCI relies on the semantic information provide by VMI and thus is vulnerable to the VMI subversion attacks (e.g., DKSM [7]). InkTag, AppShield and AppSec require the protected binaries to issue a hypercall at the startup, which triggers the integrity verification performed in the VMM.

TOCTTOU consistency: Once a code page has been verified, it should not be changed any more. Patagonix and HIMA use shadow page table to translate the GVA to HPA while AppSec and TF-BIV adopt EPT to translate the GPA to HPA. They achieve full TOCTTOU consistency as the VMM configures the verified pages non-writable based on the mechanisms provided by shadow page table or EPT. En-ACCI fails to provide TOCTTOU consistency as it doesn't provide any protection for the verified code pages. InkTag and AppShield isolate the critical application from the untrusted guest OS by adopting two EPTs, one for the critical application and one for all the others, while the code is verified before being executed in the isolated environment.

Transparency: The to-be-protected binaries and VM OS should not need any modification. In Patagonix, En-ACCI and TF-BIV, the integrity checking is transparent to the guest OS and the target programs because the solution is deployed directly in the VMM without any modification (e.g., hooks, drivers, libraries) in the guest VM. For Patagonix and TF-BIV, the integrity verification is triggered and performed in VMM directly if the permission on the physical frames (i.e., non-executable) is violated. En-ACCI invokes the verification in VMM directly once the VMM captures a cryptographic service invocation. HIMA doesn't require any modification to most OS kernels which provide native support of W⊕X, while others (e.g., Linux kernel v2.6.18) need to be patched with PaX [76]. InkTag and AppShield require the modification and redeployment of the specified applications for invoking a set of hypercalls to trigger the protection provided by VMM. AppSec requires the VM to deploy a safe loader, which distinguishes S-processes from others and invokes hypercalls to trigger the integrity verification for S-processes.

Fine-grained Verification: In TF-BIV, InkTag, AppShield and AppSec, only the specified processes are thoroughly checked while the others may be installed and updated arbitrarily. The details for identifying the S-process in TF-BIV are provided in Section 4. In InkTag and AppShield, the programs declare they are sensitive by invoking the corresponding hypercall actively. In AppSec, the safe loader, running in VM, distinguishes the S-process from others. Patagonix and HIMA verify all the binaries before execution, attempting to provide a safe environment without rootkits. En-ACCI fails to perform thorough verification for S-process, as it doesn't verify the dependent libraries.

6 THE APPLICATION IN CLOUD-BASED CRYPTOGRAPHIC SERVICES

We integrate TF-BIV with a cloud-based cryptographic service. TF-BIV provides integrity verification for the programs that need access to cryptographic keys stored outside of the VM.

Motivation. Cloud-based cryptographic service is a sensitive service, the abuse of which may introduce significant damages. For example, the abuse of the cryptographic signing service may result in an illegal transaction. For the cloud-based cryptographic service, usually the cryptographic algorithms are semantically secure and the cryptographic key is strong enough. However, the authentication and authorization mechanisms are often too weak to prevent the adversary from abusing the cryptographic service. As described in Section 1, the existing authentication mechanism of the cloud-based cryptographic service is based on identification and passwords, and the service is authorized to the entire VM once the authentication passed, even only one or a small number of processes running in the VM actually need to invoke the service. Therefore, the adversary, who obtains the leaked identification and passwords [13, 75], may exploit the vulnerability of the VM OS or any application, to illegally access the cryptographic computations for his purpose.

6.1 Integration into Cloud-based Service

When using TF-BIV to protect cloud-based cryptographic service, the tenants need to specify a list of binaries that are authorized to invoke the cloud-based cryptographic service. TF-BIV analyzes the dependency of the binary, calculates the reference hash values for each code page of the authorized binaries and the dependency (i.e., the OS kernel, LKMs and shared libraries), and stores the reference hash values in the database.

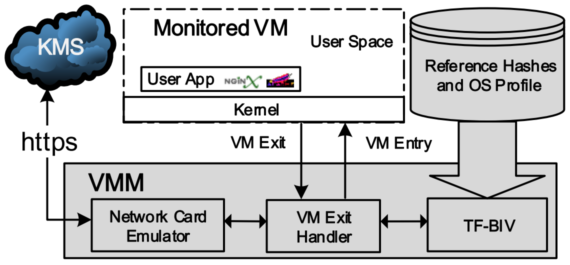

Figure 3: TF-BIV applied in cloud-based cryptographic service.

An overview of the integration of TF-BIV in the authorization of cloud-based cryptographic service is provided in Figure 3. Firstly, TF-BIV transparently identifies and verifies the integrity of the S-processes (and the dependencies) in the corresponding VM exit handlers as described in Section 4, and maintains a list of verified S-processes. Then, before sending the cryptographic service request to the remote cryptographic service provider and returning the response to the requester, the agent deployed in the VMM identifies the process that issues the cryptographic request, and transmits the request and response only if the process is in the S-process list.

As the real-world cloud-based cryptographic services [2, 20, 34, 55] usually provides the service through HTTPS connections, we deploy the agent in the network card emulator. When the IP address and port indicate that an HTTPS connection to the cloud-based cryptographic service is established or being used, the process corresponding to this HTTPS connection needs to be located to enforce the process-level authorization. If the process is not in the list of verified S-processes, the HTTPS connection will be dropped directly.

The semantics information of the guest VM OS is required to identify the process corresponding to the HTTPS connection. In addition to the assumption described in Section 3, we assume that the profile of OS kernel (i.e., the kernel data structure and the logical addresses of the essential kernel symbols) is not tampered with during the integration. Specifically, in our implementation, the adopted kernel data structures include `task_stuct`, `mm_struct`, `files_struct`, `fdtable` and `file`, while the used system symbols are `init_task`, `socket_operation` and `socket_dentry_operation`. TF-BIV verifies the integrity of the kernel code, and may integrate OSck [35] or KI-MON [50] to protect the kernel data for the correctness of obtained semantics information.

7 IMPLEMENTATION

We have implemented a prototype of TF-BIV based on QEMU-KVM v1.7.1, and have integrated it in the process-level authorization for a cloud-based cryptographic service.

7.1 TF-BIV Implementation

TF-BIV is implemented as a KVM component. In details, TF-BIV sets MTF flag and CR3-load exiting flag in VMCS, configures the EPT entries to trigger necessary VM exits, and completes the identification and integrity verification of the processes in the corresponding VM exit handlers. All the implementation is completed with less than 1000 lines of code.

As described in Section 4, TF-BIV triggers four VM exits (CR3-load exiting, MTF exiting, EPT NW exiting, and EPT NX exiting), for tracking the S-process creation, monitoring the memory page and modification of S-processes, and verifying the integrity of the code for S-process. TF-BIV invokes the function `vmcs_write` to set the corresponding bit in `VMCS` to register each VM exit events. For the CR3-load exit event, TF-BIV sets the `Cr3-load exiting` bit, clears the `Cr3-target value` and configures the `Cr3-target count` to `0` in `VMCS`, which causes a VM exit once an instruction loads any value to the `CR3` register. In the corresponding handler, TF-BIV checks whether the process is a newly created and identifies the newly created process by comparing the code page with the reference hash values.

To monitor the memory pages for S-processes, TF-BIV invokes the functions `vmcs_read` to obtain the base physical address of S-process from the CR3 register, identifies the memory areas corresponding to the memory pages, and sets the page table pointed by CR3 value as non-writable in the handler of CR3-load exiting. Once a VM exit event occurs due to the modification of S-process's page table, TF-BIV makes the page table as writable by configuring the corresponding EPT entries and sets the MTF bit in `VMCS` to obtain the stable version of the page table before the first instruction of the newly mapped code page executed. In the handler of MTF exiting, TF-BIV clears the MTF bit and sets the memory pages as non-writable again to capture the further modification.

To register EPT violation events, TF-BIV sets the access bits on the `pte` entry in EPT. According to the `exit-qualification` obtained from `VMCS`, the corresponding functions will be invoked to handle the EPT violation events. For each loaded code page, TF-BIV sets it as non-executable by configuring the `pte` entry. In the handler, TF-BIV invokes `vmcs_read` for the GVA of page fault from the CR2 register, obtains the memory area information from `task_struct.mm`, identifies the binaries for the process, verifies the integrity according to the type of the code, and sets the code page as executable but non-writable once the integrity verification passed. In the handler of EPT NW exiting, TF-BIV sets the modified code page as non-executable again for the re-verification.

For different types of code (i.e., kernel, LKMs, S-processes, and shared libraries), various analyzers are adopted to generate the reference hash values. For S-processes and shared libraries, the analyzer obtains the `.text` content, entry point, offset and alignment from the `ELF` file directly, and adopts the hash algorithm to generate the reference hash value for each code page, which is stored with the offset information in the database. For S-processes, the code page containing the entry point will be identified to accelerate the identification of S-process. The kernel and LKM will modify the instructions for optimization based on the hardware platform, which is specified in a special section [54]. TF-BIV simulates the loading process of the kernel and LKM in the target platform, and generates the corresponding reference hash values. For LKM, the analyzer parses the binary according to the section header table and an array of `Elf64_Shdr` structures to obtain the `.text`, `.data` and `symbols` for the `init` and `core` blocks. For the relocatable addresses in each binary, TF-BIV sets them as zero in calculating the hash values.

7.2 Integration

When TF-BIV is applied to the cloud-based cryptographic service for process-level authorization, we need to find the process that issues network connection between the VM and the service provider. To achieve this, we modify the `e1000` card emulator in QEMU with less than 400 lines of code. In details, in the functions `e1000_send_packet` (sending network packets) and `e1000_receive_iov` (receiving network packets), we find the process corresponding to the TCP packets whose IP address and port are consistent with these of the service provider, and transmit the data only when the process is in the S-process list.

We find the process corresponding to the packets based on the guest OS profile, which is generated by the memory forensics tool volatility [80] and dwarf-tools. For example, in Linux, because the network connection is processed as a file, we parse the `files` of the type `files_struct` for each process (i.e., each element of the type `task_struct` in the kernel variable `tasks`) to obtain the files opened by each process. After parsing the kernel data structure `file` for each file, we check whether the file represents a network connection. This is achieved by checking the `socket_file_ops` function pointer array. We parse the `dentry` further to obtain the IP address and port, and check whether the connection is related to the cryptographic service. Finally, we obtain the base physical address through the variable `pgd`, which is a member of the data structure `mm_struct`. The obtained base physical address is used to check whether the process is authorized to invoke the cryptographic service.

8 PERFORMANCE EVALUATION

We have evaluated the performance overhead imposed by TF-BIV to the startup of guest VM and to the host CPU. We have also evaluated the introduced network delay and the performance impact when TF-BIV is integrated in the cloud-based cryptographic service. The evaluation was conducted on a Dell Optiplex 9020 PC with Intel i7-4770 CPU (3.4GHZ) and 16GB RAM. The host runs the Linux OS (kernel v3.13) with QEMU v1.7.1, while the guest VM is assigned with 4 vCPUs and 4GB RAM, and the guest OS is Linux kernel v3.13.7.

Startup: We used `Bootchart` [32], a tool for performance analysis of Linux booting process, to analyze the performance overhead introduced by TF-BIV on the startup. With `Bootchart`, we measured the startup time of the VM OS with TF-BIV deployed, and compared it with that on the native host. We have performed the measurement for 10 times. The average overhead is 1.49%, which is mainly introduced by the additional integrity verification.

SPECINT Benchmark: SPECINT 2006, comprising of a set of performance benchmarks, is used to evaluate the influence on the virtualized VM. We created an S-process which issues a cryptographic service request every 5 seconds and measured the SPECINT 2006 scores. We compared the results on three scenarios: native Linux, with TF-BIV being active and with TF-BIV being active to monitor S-process. As illustrated in Figure 4, the impact on the evaluated

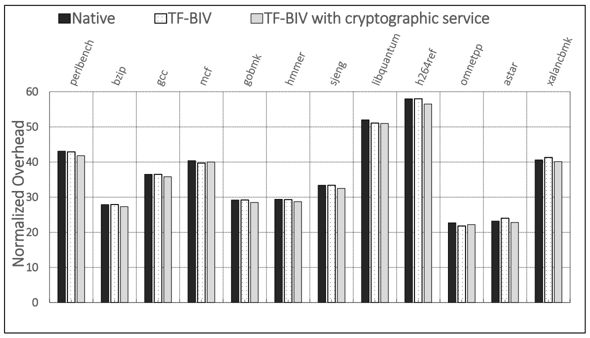

Figure 4: SPECINT 2006 perf. overhead.

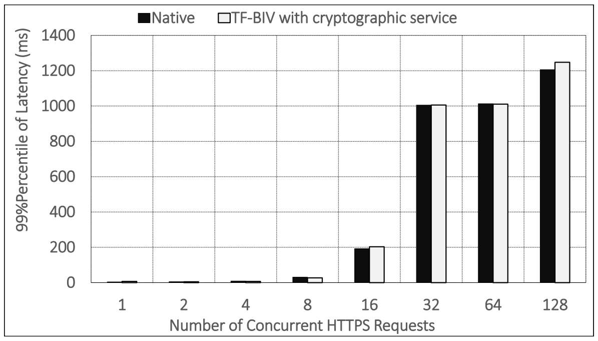

Figure 6: 99th Percentile of HTTPS latency.

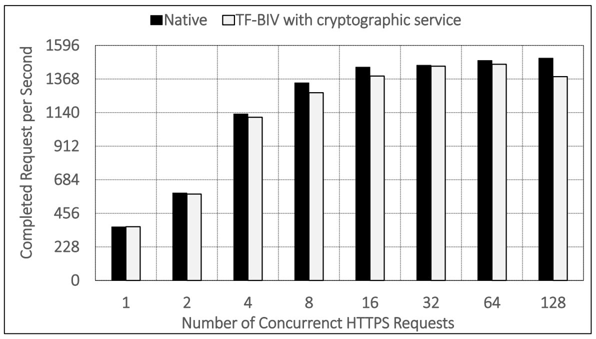

Figure 5: Speed of HTTPS service.

performance is less than 3.6%, which can be attributed to 1) identification and integrity verification of S-processes, and 2) capturing and filtering the network connection in network card emulation.

Network Performance: We adopts iPerf [43], an active measurement tool of network bandwidth, to evaluate the influence on the network introduced by TF-BIV application. In the evaluation, we used the VM with TF-BIV deployed as the client and server respectively, while the other peer was a host with Intel i5-4590 CPU (3.3GHZ) and 16GB RAM. We evaluated the overhead in the full-duplex (i.e., Dual Testing) and half-duplex (i.e., Tradeoff Testing) scenarios. As shown in Table 2, the bandwidth decreases by 3.81% when the VM (monitored by TF-BIV) works as the server in the dual testing. The reason is that TF-BIV needs to process more connections when the VM works as the server and needs to filter more packets in the dual testing as both the client and server send packets.

Table 2: Bandwidth evaluation.

Testing Mode		Native	TF-BIV
Dual Testing	C	920.67MB/s	894.00MB/s
	S	632.11MB/s	608.00MB/s
Tradeoff Testing	C	819.78MB/s	810.56MB/s
	S	1089MB/s	1086MB/s

C: VM with TF-BIV as the client, S: VM with TF-BIV as the server.

HTTPS Throughput and Latency: We have integrated TF-BIV with the cloud-based cryptographic service for process-level authorization, and evaluated the performance overhead by constructing an HTTPS service which requests the cloud-based cryptographic service for RSA-2048 decryption. In details, the RSA-2048 decryption was deployed on a separated host and provided service through HTTPS connections. We ran the Apache in the monitored VM, which requests cryptographic service for HTTPS. Therefore, `httpd` is the S-process in our experiment. We also used the Apache benchmark to evaluate the throughput and latency of HTTPS service. For each evaluation, the client constructed 10,000 HTTPS requests for a 4KB web page at different concurrency levels. As shown in Figure 5, the maximum decrease in throughput is 8.3% and the largest latency is 5.7%, which occurs at the largest concurrency level (i.e., 128). This is because the network card emulator needs to process more connections and packets (i.e., finding the process corresponding to the packets, and obtaining the IP address and port for the packets) when the concurrency level increases.

9 RELATED WORK

Various in-kernel solutions have been proposed to protect the integrity of user space programs. Integrity Measurement Architecture (IMA) [65] and the extension PRIMA [45], deployed in the Linux kernel, measure all binaries at load-time based on TPM, but fail to provide the run-time integrity protection.

Hypervisor, as layer between hardware and the OS, has been leveraged in many solutions. Patagonix [54] and HIMA [5] aim to safeguard the guest VM, but lack flexibility, because all the binaries running in the VM need to be protected. AppSec [59], AppShield [19] and InkTag [36] protect both the integrity and confidentiality of the sensitive applications even if the guest VM OS is untrusted. However, they require modification to the protected applications. Moreover, various schemes (e.g., HyperCheck [81], HyperSentry [6]) have been proposed to protect the hypervisor itself.

Many hardware features (e.g., Intel SGX [42], AMD SEV [48], ARM TrustZone [3]) have been proposed to provide the isolation for the sensitive applications [4, 8, 68]. However, these solutions [4, 8, 68] require the substantial re-engineering effort.

The adversary may hijack the control flow without injecting or modifying the binaries [10, 11, 14, 63, 70, 77]. For these attacks, various protections [21, 28, 38, 52, 67, 74] are proposed, which may be integrated with TF-BIV. Heisenbyte [74] utilizes Intel EPT for garbling the code right after it is read, to prevent code reuse attacks. PITTYAPT [28] uses the Intel PT [42] to enforce path-sensitive CFI. KVM-PT [67] extends PITTYAPT to the virtualized environment. μCFI [38] guarantees only one valid target for each invocation of an indirect control-flow transfer based on the points-to analysis on control data with the collected constraining data, to reduce the illegal jumps possibility.

Data-oriented attack manipulates the target data directly [17, 37], or constructs malicious sequences of instructions by tampering with multiple chosen data [39, 44], which alters the program's benign behavior without being detected or prevented by the mechanisms of CFI. Various data-oriented attacks have been proposed for the real-world software, such as Chrome [46, 64], Linux Kernel [26], ProFTPd [39] and Nginx [56]. As systematically analyzed in [18], the generic memory corruption defenses [27, 49, 57, 71], and the special defenses [12, 16, 26, 58, 66, 72] may be used to construct the three-level defense for the general/dedicated data-oriented attacks by avoiding exploitation of memory errors, increasing the difficulty for guessing memory layout or preventing the use of corrupted data. These schemes may be integrated with TF-BIV, to strengthen sensitive applications.

10 CONCLUSION

In this paper, we provide TF-BIV, a binary integrity verification scheme for the cloud environment, which achieves isolation, transparency, TOCTTOU consistency and fine-grained verification simultaneously. TF-BIV leverages hardware virtualization to achieve transparent and fine-grained verification, and adopts the semantic information obtained through VMI to accelerate the identification of S-process and integrity verification. TF-BIV registers VM exit events based on the hardware features (i.e., CR3-load exit, MTF, and EPT volition) to transparently capture the creation of process, identify all the dependent code of S-processes and continuously perform the integrity verification for S-processes. Moreover, TF-BIV can be easily integrated with real-world applications that need protection. To demonstrate this, We integrate TF-BIV with a real open source cloud-based cryptographic service. The evaluation demonstrates that the performance overhead introduced by TF-BIV is modest – we observed less than 3.6% overhead in CPU benchmarking, about 3.81% network overhead, and about 8.3% degradation of the throughput in cloud-based cryptographic service.

ACKNOWLEDGMENT

This work is supported partially by 973 Program of China (Grant No. 2014CB340603), Informatization Project of Chinese Academy of Sciences (Grant No. XXH13507-01), Cyber Security Program of National Key RD Plan of China (Grant No. 2017YFB0802100), National Natural Science Foundation of China (Grant No. 61772518), National Science Foundation (Grant CNS-1422206 and DGE-1565570), and National Security Agency (NSA) Science of Security (SoS) initiative.

REFERENCES

[1] Amazon. 2018. *AMAZON AWS CloudHSM.* Retrieved May 27, 2019 from https://amazonaws-china.com/cloudhsm/
[2] Amazon. 2018. *AMAZON AWS Key Management Service KMS.* Retrieved May 27, 2019 from https://amazonaws-china.com/kms/
[3] ARM 2009. *ARM security technology: Building a secure system using TrustZone technology.* ARM.
[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 689–703.
[5] Ahmed M. Azab, Peng Ning, Emre Can Sezer, and Xiaolan Zhang. 2009. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Twenty-Fifth Annual Computer Security Applications Conference, ACSAC 2009, Honolulu, Hawaii, USA, 7-11 December 2009.* 461–470.
[6] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. 2010. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010.* 38–49.
[7] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. 2010. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010.* 82–91.
[8] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26.
[9] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. 2014. Leveraging Virtual Machine Introspection for Hot-Hardening of Arbitrary Cloud-User Applications. In *6th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud '14, Philadelphia, PA, USA, June 17-18, 2014.*
[10] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. 2018. Smashing the Stack Protector for Fun and Profit. In *ICT Systems Security and Privacy Protection - 33rd IFIP TC 11 International Conference, SEC 2018, Held at the 24th IFIP World Computer Congress, WCC 2018, Poznan, Poland, September 18-20, 2018, Proceedings.* 293–306.
[11] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011.* 30–40.
[12] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA.* 147–160.
[13] Rahul Chatterjee, Joseph Bonneau, Ari Juels, and Thomas Ristenpart. 2015. Cracking-Resistant Password Vaults Using Natural Language Encoders. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* 481–498.
[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010.* 559–572.
[15] Ping Chen, Dongyan Xu, and Bing Mao. 2012. CloudER: a framework for automatic software vulnerability location and patching in the cloud. In *7th ACM Symposium on Information, Compuer and Communications Security, ASIACCS '12, Seoul, Korea, May 2-4, 2012.* 50.
[16] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017.* 167–178.
[17] Shuo Chen, Jun Xu, and Emre Can Sezer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005.*
[18] Long Cheng, Hans Liljestrand, Thomas Nyman, Yu Tsung Lee, Danfeng Yao, Trent Jaeger, and N. Asokan. 2019. Exploitation Techniques and Defenses for Data-Oriented Attacks. *CoRR* (2019).
[19] Yueqiang Cheng, Xuhua Ding, and Robert H. Deng. 2015. Efficient Virtualization-Based Application Protection Against Untrusted Operating System. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015.* 345–356.
[20] Alibaba Cloud. 2019. *Aliyun cryption service.* Retrieved May 27, 2019 from https://www.aliyun.com/product/kms
[21] Crispan Cowan. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998.*
[22] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. 2001. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA.*

[23] CVE20188492 2019. Common Vulnerabilities and Exposures. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8492.

[24] CVE20190247 2019. Common Vulnerabilities and Exposures. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-0247.

[25] CVE20196250 2019. Common Vulnerabilities and Exposures. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6250.

[26] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017.*

[27] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008.* 103–114.

[28] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* 131–148.

[29] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon T. Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA.* 297–312.

[30] FIPS-140-2 2019. Security Requirements for Cryptographic Modules. https://www.nist.gov/publications/.

[31] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA.* San Diego, California.

[32] Inc. Gentoo Foundation. 2019. *Boot Process Performance Visualization.* Retrieved May 27, 2019 from http://www.bootchart.org/

[33] Google. 2018. *GOOGLE Cloud HSM.* Retrieved May 27, 2019 from https://cloud.google.com/hsm/

[34] Google. 2018. *GOOGLE CLOUD KEY MANAGEMENT SERVICE.* Retrieved May 27, 2019 from https://cloud.google.com/kms/

[35] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring operating system kernel integrity with OSck. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011.* 279–290.

[36] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: secure applications on an untrusted operating system. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013.* 265–278.

[37] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-Oriented Exploits. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.* 177–192.

[38] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1470–1486.

[39] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016.* 969–986.

[40] Zhichao Hua, Dong Du, Yubin Xia, Haibo Chen, and Binyu Zang. 2018. EPTI: Efficient Defence against Meltdown Attack for Unpatched VMs. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.* 255–266.

[41] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive* 2015 (2015), 898.

[42] Intel Corporation 2019. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel Corporation.

[43] iPerf 2017. *The ultimate speed test tool for TCP, UDP and SCTP.* Retrieved May 27, 2019 from https://iperf.fr

[44] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 1868–1882.

[45] Trent Jaeger, Reiner Sailer, and Umesh Shankar. 2006. PRIMA: policy-reduced integrity measurement architecture. In *11th ACM Symposium on Access Control Models and Technologies, SACMAT 2006, Lake Tahoe, California, USA, June 7-9, 2006, Proceedings.* 19–28.

[46] Yaoqi Jia, Zheng Leong Chua, Hong Hu, Shuo Chen, Prateek Saxena, and Zhenkai Liang. 2016. "The Web/Local" Boundary Is Fuzzy: A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* 791–804.

[47] Fangjie Jiang, Quanwei Cai, Le Guan, and Jingqiang Lin. 2018. Enforcing Access Controls for the Cryptographic Cloud Service Invocation Based on Virtual Machine Introspection. In *Information Security - 21st International Conference, ISC 2018, Guildford, UK, September 9-12, 2018, Proceedings.* 213–230.

[48] Kaplan, David 2016. AMD x86 Memory Encryption Technologies.

[49] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 147–163.

[50] Hojoon Lee, Hyungon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. 2013. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013.* 511–526.

[51] Kyung-suk Lhee and Steve J. Chapin. 2003. Buffer overflow and format string overflow vulnerabilities. *Softw., Pract. Exper.* 33, 5 (2003), 423–460.

[52] Jinku Li, Zhi Wang, Tyler K. Bletsch, Deepa Srinivasan, Michael C. Grace, and Xuxian Jiang. 2011. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Trans. Information Forensics and Security* 6, 4 (2011), 1404–1417.

[53] Song Li and Scott Wu. 2018. Your Device and Your Power, My Bitcoin. In *Blockchain - ICBC 2018 - First International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings.* 285–292.

[54] Lionel Litty, H. Andrés Lagar-Cavilla, and David Lie. 2008. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA.* 243–258.

[55] Microsoft. 2018. *Microsoft Key Vault.* Retrieved May 27, 2019 from https://www.azure.cn/home/features/key-vault/

[56] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Z. Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2018. Security Risks in Asynchronous Web Servers: When Performance Optimizations Amplify the Impact of Data-Oriented Attacks. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018.* 167–182.

[57] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009.* 245–258.

[58] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, and Ahmad-Reza Sadeghi. 2017. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement. *CoRR* (2017).

[59] Jianbao Ren, Yong Qi, Yue-hua Dai, Xiaoguang Wang, and Yi Shi. 2015. AppSec: A Safe Execution Environment for Security Sensitive Applications. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Istanbul, Turkey, March 14-15, 2015.* 187–199.

[60] ISEC Security Research. [n.d.]. *knox-implementation of non-executable page protection mechanism.* http://isec.pl/projects/knox/knox.html

[61] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Recent Advances in Intrusion Detection, 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings.* 1–20.

[62] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 2009. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009.* 199–212.

[63] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1 (2012), 2:1–2:34.

[64] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. 2017. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017.* 366–381.

[65] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA.* 223–238.

[66] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin G. Zorn. 2011. Modular Protections against Non-control Data Attacks. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011.* 131–145.

[67] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* 167–182.

[68] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 38–54.

[69] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*. 335–350.

[70] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*. 552–561.

[71] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004, Washington, DC, USA, October 25-29, 2004*. 298–307.

[72] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 1–17.

[73] Starzetz. [n.d.]. *RSX*. Retrieved May 27, 2019 from http://www.starzetz.com/software/rsx/

[74] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. 256–267.

[75] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. 2012. CleanOS: Limiting Mobile Data Exposure with Idle Eviction. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. 77–91.

[76] The PaX Team. 2013. *PaX*. Retrieved May 27, 2019 from https://pax.grsecurity.net/

[77] Minh Tran, Mark Etheridge, Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*. 121–141.

[78] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. 430–444.

[79] Vendicator. [n.d.]. *Stack Shield: A Stack Smashing Tecnique protection tool for Linux*. Retrieved May 27, 2019 from http://www.angel?re.com/sk/stackshield

[80] Volatility 2015. The Volatility Framework. https://code.google.com/archive/p/volatility/.

[81] Jiang Wang, Angelos Stavrou, and Anup K. Ghosh. 2010. HyperCheck: A Hardware-Assisted Integrity Monitor. In *Recent Advances in Intrusion Detection, 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*. 158–177.

[82] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. 380–395.

[83] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/usenixsecurity19/presentation/wu-wei

[84] Su Zhang. 2012. *Deep-diving into an easily-overlooked threat: Inter-VM attacks*. Technical Report. Technical Report. Manhattan, Kansas: Kansas State University.

[85] Shuhui Zhang, Xiangxu Meng, Lianhai Wang, Lijuan Xu, and Xiaohui Han. 2018. Secure Virtualization Environment Based on Advanced Memory Introspection. *Security and Communication Networks* 2018 (2018), 9410278:1–9410278:16.