Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools and methods for systems and binary analysis on modern platforms.

Marcus Botacin, University of Campinas
Paulo Lício de Geus, University of Campinas
André Grégio, Federal University of Paraná

Malicious software, a threat users face on a daily basis, have evolved from simple bankers based on social engineering to advanced persistent threats (APTs). Recent research and discoveries reveal that malware developers have been using a wide range of anti-analysis and evasion techniques, in-memory attacks and system subversion, including BIOS and hypervisors. In addition, code-reuse attacks like Returned Oriented Programming (ROP) emerge as highly-potential remote code execution threats. To counteract the broadness of malicious codes, distinct techniques and tools have been proposed, such as transparent malware tracers, system-wide debuggers, live forensics tools and isolated execution rings. In this work, we present a survey on state-of-the-art techniques that detect, mitigate and analyze the aforementioned attacks. We show approaches based on Hardware Virtual Machines introspection (HVM), System Management Mode (SMM) instrumentation, Hardware Performance Counters (HPCs), isolated rings (e.g., Software Guard eXtensions), as well as others based on external hardware. We also discuss upcoming threats based on the very same technologies used for defense. Our main goal is to provide the reader with a broader, more comprehensive understanding of recently-surfaced tools and techniques aiming at binary analysis for modern platforms.

CCS Concepts:• **Security and privacy** → **Malware and its mitigation; Software reverse engineering;** *Information flow control;*

General Terms: Binary Analysis, Malware, Security, HVM, SMM, Introspection

Additional Key Words and Phrases: Binary Analysis, Malware, Security, HVM, SMM, Introspection

ACM Reference Format:
Marcus Botacin, Paulo Lício de Geus, André Grégio, 2017. Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools and methods for systems and binary analysis on modern platforms. *ACM Comput. Surv.* 0, 0, Article 0 (To appear), 41 pages.
DOI: 0

1. INTRODUCTION

Keeping current systems secure is a task that is both critical and hard to achieve. Current attacks go from remote code execution, such as Return Oriented Programming (ROP), to targeted, sophisticate malware attacks. Reported data shows malware dissemination alarming rates: figures exceeding 60 thousand new samples per day [Ashford 2010]; mobile devices growing as targets [Townsend 2016]; losses amounting to around 200 million dollars in the first quarter of 2016 [Fitzpatrick and Griffin 2016],

This work was supported by the Brazilian National Council of Technological and Scientific Development (CNPq, Universal 14/2014, process 444487/2014-0) and the Coordination for the Improvement of Higher Education Personnel (CAPES).
Author's addresses: Marcus Botacin and Paulo Lício de Geus, Institute of Computing, University of Campinas; André Grégio, Department of Informatics, Federal University of Paraná.

and so on. To handle security incidents generated by these threats, researchers have proposed plenty of tools to detect/prevent (e.g., Intrusion Detection/Prevention Systems, packet filters), mitigate (antivirus, vaccines) and analyze (sandboxes) malicious code. Malware in general have been increasing both in numbers and in complexity, bypassing filters through polymorphism, evading sandboxes through virtual environment detection and even subverting whole system operation by taking over control of hypervisors. Therefore, new approaches are required to address this threat.

The bulk of novel defensive approaches is based on techniques that stealthily/transparently acquire data from target systems, notably through System Management Mode (SMM) and Hardware-Assisted Introspection. Such analytical approaches are hardware-supported and run in the most privileged ring, as is desirable for reliable experiments [Rossow et al. 2012]. As a result, a fine-grained view of analyzed subjects is possible, even if they make use of evasive actions. In addition, other approaches have emerged, such as lightweight, performance-counter-based ones and isolated execution rings, thus preventing payload tampering. Many of the proposed approaches are still sparsely used, while threats to them are constantly evolving. This leads to a dangerous scenario of stealthy and Operating System-independent threats.

Therefore, it is important to provide an overview of the current situation, as well as to present malicious applications built with these underlying techniques and related work that might guide novel research in the field. Hence, this survey can be seen as a comprehensive review of current, state-of-the-art techniques for system security based on recently established and existing hardware-assisted solutions, which aims to better position them in the security context. In summary, our paper presents the following:

Motivation: many distinct security-related solutions have been arising, noticeably hardware-supported ones. However, a clear scenario is not established yet, making it harder to identify which solution best fits each usage scenario. Therefore, we propose to shed some light on the advantages and limitations of these solutions.

Contribution: our main contribution is to define a panorama of current state-of-the-art monitoring tools, in order to allow researchers to identify existing development gaps and so to better position their solutions.

Target: we are mostly focused on x86-based systems, since it is the most deployed and targeted architecture, which allows us to understand security solutions evolution over time. We also target, whenever possible, ARM-based systems, since these are playing a major role in mobile devices and are also quite present in the IoT world.

Article's outline: in Section 2, we show work closer to ours; in Section 3, we discuss the properties and requirements for modern security solutions; in Section 4, we present a background on monitoring technologies used to deploy modern security solutions; in Section 5 we show common challenges faced when implementing security solutions using the presented technologies; in Section 6, we introduce security solutions implemented using the previously mentioned technologies; in Section 7, we summarize existing development gaps and research opportunities; finally, in Section 8, we draw our conclusions.

2. RELATED WORK

In this section, we discuss other surveys on the same topic as our paper. Compared to these other works, we try to survey the entire field of malware analysis, while others focus only on works that are closely related to their own approaches.

This way, while referring to ROP attacks for instance, we are not bound to minor details nor to describe known policies, such as `Write`$\oplus$`Execute` [Roemer et al. 2012]. Instead, we try to figure out how such policies can be deployed on modern systems. As for malware, our aim is not to rehash static analysis limitations [Moser et al. 2007] or the exhaustive list of existing anti-analysis tricks they employ [Marpaung et al. 2012;

Gao et al. 2014], but to present distinct technologies and methods of implementing countermeasures, and to discuss hardware issues as [Pék et al. 2013] did.

In this sense, our work can be considered as an updated, complementary version of the survey on dynamic analysis [Egele et al. 2008], which presented many system monitoring implementations, such as DLL injection, SSDT hooking, and so on. However, we target implementations enabled by modern systems, such as those assisted by hardware. In this context, the closest work to ours is about Trusted Execution Environments (TEEs) [Zhang and Zhang 2016]. We begin from where they stopped and cover the topics in a broader way.

3. SECURITY SOLUTIONS REQUIREMENTS

In this section, we provide an overview of properties and requirements that are desirable for security solutions, such as malware analysis, forensics and detection systems.

3.1. Transparency

An analysis system should be evasion-resistant, since malware samples often try to evade analysis environments [Chen et al. 2008a]. Given that most analysis systems run on top of emulators and/or software-based VMs, malware can detect them by testing instruction behavior, which often differs from the ones presented in a real CPU. [Martignoni et al. 2009] describes a method of fuzzy-testing a CPU emulator (called Red Pill) that generates tests for emulator identification (extended in [Shi et al. 2014]). In addition, [Paleari et al. 2009] developed a way of automatically generating red pills.

Although there are research efforts to overcome these challenges, proposed solutions are very costly, since they either require execution in multiple environments, such as BareCloud [Kirat et al. 2014] and Splitmal [Balzarotti et al. 2010], or require a physical machine, such as Barebox [Kirat et al. 2011]. A more viable approach requires keeping simpler hardware, which in turn demands a more transparent set of monitoring techniques. In [Dinaburg et al. 2008a]'s definition, transparency is achieved by meeting the following criteria: (i) **higher privilege**—the analyzer has to be more privileged than its subject; (ii) **no non-privileged side effects**—any instruction that induces side effects should be handled by an exception able to hide it; (iii) **identical basic instruction semantics**—each executed instruction needs to have the same effect and lead to the same next instruction; (iv) **transparent exception handling**—given an exception on a given i^{th} instruction, the exception handler must return to the $i+1^{th}$ instruction; (v) **identical measurement of time**—the measurement of time has to be identical within and without the analyzer, but as this requirement is hard to fulfill (exception handling has not constant time), small differences are tolerable. Most of these criteria are naturally met by bare-metal systems, such as those based on SMM and HPC-enabled ones. In addition, HVM may also meet them, since code is allowed to run on the native processor.

3.2. Live loading

Forensic solutions should be able to load on demand, only collecting data when requested. Thus, no privacy issue would be raised nor performance penalties would be imposed due to constant monitoring. While built-in solutions are not able to meet this criteria because they need to be pre-launched, HVM has a late-launch ability.

3.3. Performance

Real time security solutions should not degrade system performance in order to keep the original application "usable". In this sense, ordinary monitoring systems such as software-VMs may be unsuitable, but hardware support improves performance [Opsahl 2013].

First-generation performance counters (e.g. LBR and BTS) stored captured data on a per-branch basis, while the modern processor tracer (PT) uses an efficient encoding that stores target addresses only for taken branches. Moreover, memory-based systems can efficiently provide data by relying on snapshots. By using memory management support, system pages can be flagged as not present, thus causing page faults when written. In this mode, only the affected pages are supplied to the monitor, since the others remain unchanged. This mode may be viewed as derived from the copy-on-write Unix concept, often named as dump-on-write. Such memory support is available on modern MMU systems.

3.4. Synchronicity

An important project decision for system-integrity checking solutions is how data will be collected—on a periodic, snapshot basis or in an event-driven one. Most approaches have a significant drawback regarding their snapshot feature, making them prone to timing attacks. Also known as transient attacks, timing attacks allow for an attacker to remain stealthy by executing malicious activities during the interval between two snapshots. As a result, modern monitoring approaches must be implemented in an event-driven way to remain unaffected against timing attacks. However, implementing an event-aware monitor requires more introspection, since translating signals into events demands deeper understanding of the system. In addition, sustained monitoring may create performance bottlenecks for bus snooping approaches.

3.5. Development effort and trusted code base

Any security solution must take into account its development effort and the required Trusted Code Base (TCB), i.e. the amount of code that must be trusted *a priori*. Relying on existing libraries and features reduces the development cost, but increases the trusted code base. For instance, a kernel driver may rely on existing O.S. support but must restrict its threat model to userland, since the whole kernel must be trusted. An HVM-based solution, in turn, is able to monitor both kernel and userland, but requires writing a costly hypervisor monitor. The same reasoning is valid for SMM-based solutions, but in this case, as they operate in the BIOS, even NIC drivers require implementation within this level to support network access.

3.6. Build once, run always

A desired feature for a detection system is that, once developed, it could run without requiring refactoring/recompilation. However, as signature-based techniques are very popular, solutions based on such checking do not often meet this criteria. For example, many solutions are built on top of Dynamic Binary Instrumentation (DBI) tools, which allows for a fine-grained analysis at the instruction level. DBI's use in security became popular due to DynamoRIO [DynamoRIO 2001] and PIN [Intel 2015].

As a downside, execution in a DBI environment may be detected in many ways, such as by executing an instruction whose translation turns into an abnormal behavior, and timing and code-cache attacks [Polino et al. 2017]. VAMPIRE [Vasudevan and Yerraballi 2005] and SPIKE [Vasudevan and Yerraballi 2006b] attempt to solve this by allowing for stealthy fine-grained tools such as Cobra [Vasudevan and Yerraballi 2006a], thus being able to translate some known failure-prone instructions into surely successful ones.

However, all known fail-prone instructions require translation, which is not only impractical [Kang et al. 2009], but also prone to evasion by malware that employ an as yet unknown "evasion trick", forcing a recompilation/reinstrumentation to be detected. To handle modern threats, the monitoring system should be able to natively handle such cases (e.g. solutions based on HVM or SMM meet this criteria).

3.7. System-wide view

Each threat model imposes a distinct monitoring level requirement. While application misbehavior detection may be limited to userland, forensic procedures may require a system-wide view covering the kernel and even the BIOS. So far, the most used technique for system-wide monitoring has been traditional virtual machine introspection (VMI), such as Anubis [Bayer et al. 2006; ISECLAB 2010] on top of QEMU [Bellard 2005] for malware analysis, Danubis [Neugschwandtner et al. 2010] version for drivers, or Bitblaze [Song et al. 2008] and Virtice [Quynh and Suzaki 2010] built on top of TEMU. In fact, VMI has become popular to the point of having automatic instrumentation tools, such as LibVMI [LibVMI 2015]. However, modern systems have become more complex: most native software is already virtualized; the BIOS runs critical code; DMA attacks are widespread; and even the chipset is able to control the processor. In this scenario, a modern security solution must consider these factors in its threat model and properly handle such events. In this line of thought, solutions based on SMM are able to analyze userland, kernel and even the hypervisor.

3.8. Abstraction levels and the semantic gap

Monitoring solutions can be placed on many levels along a modern computer system stack, providing a solution with a distinct system view: a kernel driver may interpret a given pointer as a function; hypervisors may understand the same pointer as a `CALL` instruction; SMM code may look to that and see only bytes. Views on each level are named abstraction levels, and those required by the solutions discussed in this paper (shown in Figure 1) are presented in Figure 2.

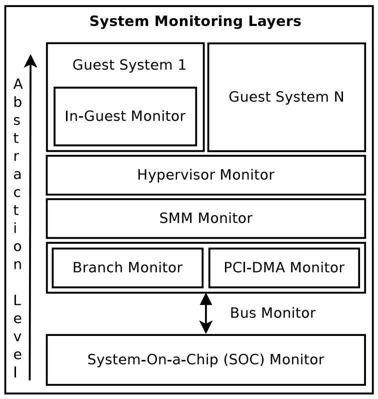

Fig. 1: Placement of distinct monitoring techniques.

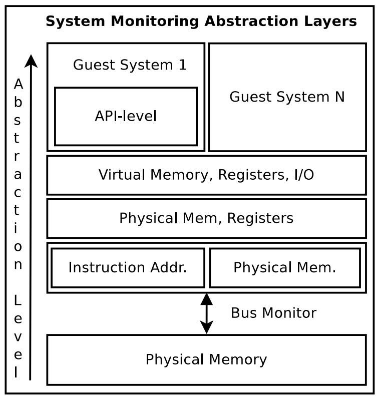

Fig. 2: Abstraction levels for distinct monitoring techniques.

For humans, byte information is not very meaningful whereas function names are easier to interpret. Abstraction levels are classified according to their closeness to human interpretation: the higher the level, the closer the ability to be interpreted. On the one hand, it is better that the monitor is far from the monitored object, since it is less prone to subversion. On the other hand, it incurs in more distance from human-readable information. This distance is named semantic gap, and the data collected in a given ring must be enriched with additional data in order to bridge such gap, a procedure named introspection.

Battle of the rings. Modern processors isolate distinct classes of applications by hardware-imposed privilege limitations. The privilege levels in the x86 architecture are known as rings, which are directly related to abstraction levels: userland applications run on `ring 3`; the kernel runs on `ring 0`; other rings were aimed to be used by dynamic libraries and drivers, but are not used in practice. Recently, as new monitoring mechanisms have been proposed, new ring names were christened in order to highlight the higher privileges they impose. This way, HVM became known as `ring -1`, since it is more privileged than the kernel's and userland's, and SMM became known as `ring -2`, since it can monitor even hypervisors. Currently, a more privileged system mode known as `ring -3`—the Management Engine (ME)—has been employed to monitor SMM.

Semantic Gap and Introspection. Semantic gap bridging may be performed in many ways, each exhibiting advantages and limitations. As presented in [More and Tapaswi 2014], the techniques may be classified in categories like memory, I/O, syscall and process. **Memory introspection** is based on memory views obtained, for instance, from dumps and consists in interpreting control blocks in memory, such as process structures. Besides being used in virtual machine introspection, it is largely used in malware scanner solutions, which search for malicious patterns in system memory. One way to implement this technique is by using shadow memory pages. **I/O introspection** uses a similar technique, but applies it to I/O; a straightforward application is to enforce I/O integrity. **Syscall introspection** is focused on context information, such as caller and arguments. Since a process is a high-level construct, its information is not available when monitoring at lower levels (e.g. processor and hypervisor). To identify the calling process, one might use the CR3 register, as it is the page table pointer and is unique for each process. In addition, context switches may also be identified, since CR3 changes accordingly; on HVM-based systems, the CR3 register may be monitored through `VM-EXITS`. When handling syscalls, knowledge about other registers, such as `eax`, helps to identify the target syscall. **Process introspection** is focused on given, specific process information, such as accessed resources, called functions etc. The first proposed mechanisms to implement it relied on hooking, which is invasive. Recently, Botacin et al. [Botacin et al. 2018] presented a process introspection mechanism for branch-monitor-collected data with no hooks. The branch instruction addresses are translated to function call names based on previous enumeration of loaded libraries and their respective function offsets.

However, developing an introspection technique for semantic gap bridging is still a hard task, since it requires OS internals knowledge and manual work. Despite the automated techniques proposed in the literature [Fu and Lin 2013; Schneider et al. 2011; Saberi et al. 2014; Dolan-Gavitt et al. 2011], some tasks are still unsolvable, since there are `one-to-many` mappings, thus making it harder to infer which high-level construction a given low-level one refers to. A variating semantic gap problem arises when a monitoring solution aims to inspect a ring positioned more than one ring above it, which is called nested/coupled semantic gap. In this case, semantic gap bridging techniques must be repeatedly applied in order to retrieve views from each level. This scenario appears in practice when the SMM mode wants to inspect the code running inside a hypervisor.

3.9. Placement and warning ability

An important decision for a monitoring system is the monitor placement. Internal monitors, such as the ones using existing hardware features, are easy to deploy and have a smaller semantic gap to be bridged but are, however, more prone to be subverted. External monitors, such as a bus-connected System-on-a-Chip (SoC), are tamper-proof, but implementing them may require hardware design knowledge. In addition, external hardware approaches have the significant disadvantage of being passive tools, leaving no possibility of blocking threats. Hence, researchers have yet to propose ways to block policy violations.

3.10. Integrity and tamper protection

A secure system should keep its own integrity for obvious reasons: it may be turned unable to detect attacks; compromised forensic solutions do not meet legal requirements and provide wrong information etc. Ensuring integrity, however, poses challenges (e.g. the required TCB) and imposes limitations. Security solutions must also spend time checking their own integrity (e.g. calculating memory hashes) apart from just monitoring a target system. Another concern is untrusted media: many solutions have to implement their own network packet checkers to prevent attacks against the monitoring solution, thus invalidating collected data. In addition, as some solutions are intended to be loaded onto compromised systems, their loading process requires verification. An example of a modern checking technique is to implement a lie detector—a system which collects the same data from multiple sources and compares them to check for tampering at any point. One way to implement it is through collecting data from both inside and outside a running VM.

3.11. Compatibility and Integration

Security solutions are often built on top of preceeding ones to benefit from available knowledge. Thus, despite not being a requirement, legacy support is a desirable feature, since it allows integrating existing solutions to newly developed ones. For instance, many solutions presented in this work, noticeably the debuggers, are integrated with GDB.

4. BACKGROUND ON SECURITY MONITORING TECHNOLOGIES

We present a background and an overview of the most recent technologies used to deploy security solutions. We also present how each one of them meet the previously mentioned desirable features/requirements.

4.1. Hardware Virtual Machines

The x86 and x86-64 CPUs have three operating modes: *Protected Mode*, which is the processor's native mode; *Real-Address Mode*, which extends the previous mode; *System Management Mode*, detailed in further sections. Virtualization instructions introduced by Intel VT-x [Intel 2013] and AMD-v/SVM [AMD 2013] platforms add new operating modes, extending the CPU instruction set. In this section, we cover the concepts of HVM operation focused on the x86-64 architecture, with implementations for both platforms. In these, the two added operating modes—*root* and *non-root*—are associated with hypervisors and guest machines, respectively. Transitions from non-root to root modes are known as *VM-EXIT*, which work like an exception/trap able to dynamically configure the set of spanning actions. After handling these events, the execution is resumed through *VM-RESUME* or *VM-ENTRY*. Figure 3 illustrates a basic scheme of these new modes and events. During transitions, instrumented hypervisors can log the exit reasons as well as perform data collection, since CPU registers are directly accessible.

An important change that comes with using virtualization instructions is in the memory controller, since HVMs implement the double address translation mechanism. In a traditional hypervisor, guest virtual addresses are translated into guest physical ones (same as the host physical addresses). Intel and AMD deployed techniques called Extended Page Table (EPT) and nested page table (NPT), respectively, that add an additional translation layer. On these systems, guest virtual addresses are translated into guest physical ones, but contrary to the previous implementations they are further

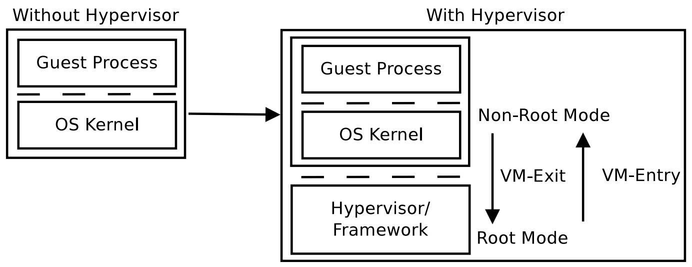

Fig. 3: VM operating layers, new modes and special instructions.

translated before getting to the host's physical address lines. This process can be seen in Figure 4. Knowledge of this mechanism is important, as the second translation level could be instrumented to monitor memory accesses through translation-faults. However, such memory monitoring is not broad enough to cover the system as a whole, given that these are CPU memory accesses; Direct Memory Accesses (DMA) may also happen, and those are external to the CPU. DMA monitoring is enabled by another mechanism called IOMMU that intercepts Input/Output (I/O) actions. More robust monitoring may be accomplished by following both approaches.

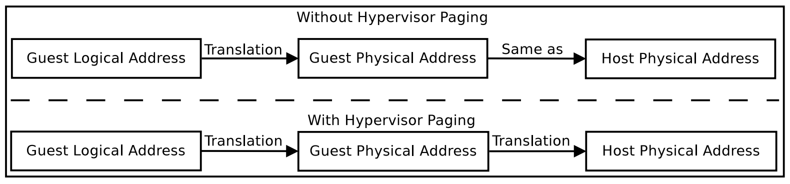

Fig. 4: VM memory operation scheme with and without hypervisor paging.

Despite many instrumentation features enabled by HVM, the main advantage is the ability of running code directly on the processor, without the need for instruction translation. This is particularly desirable for malware analysis systems, and often a drawback of DBI-based systems subject to instruction translation side-effects. Since this type of analysis system is not susceptible to CPU emulation bugs, malware running on it would not be able to rely on execution side effects, ones that could help identify whether it is running inside a virtual environment or not. Therefore, a system like that achieves a certain degree of transparency.

A Hardware-assisted Virtual Machine (HVM) is configured through special control structures, named Virtual Machine Control Structure (VMCS) and Virtual Machine Control Blocks (VMCB) on Intel and AMD systems, respectively. These blocks include the initial system state, memory allocation and VM-EXITS configuration. Their advantage resides in the system not requiring to be booted up in a virtualized environment: it may be conveniently moved to another at runtime *(late launch)*. Late launch broadens the options for security analysts to perform live forensics, since it does not require reboot or shutdown. In late launch, the initialization blocks are set to the current system state, but it does require a driver to set specific registers at a privileged execution level. When aiming to implement a security framework based on HVM, the usual approach is to instrument the hypervisor layer in root mode to collect information from non-root mode, and then send it to an external client. In this case, the abstraction is similar to the one used in operating systems: the VM monitor runs in kernelspace while the analyzer runs in userspace. Even though the external client is stealthy for malware, the hypervisor itself is not. One should notice that the malicious code could map the hypervisor's physical memory as its own, and then launch an attack from within. A solution for that issue is to put the instrumented framework in a page where malware cannot access, which is done through the newly presented memory mechanism and its fault handler. The internal driver that loads the hypervisor can also be detected. To counter that, a tool may employ rootkit techniques to hide itself from the system, or else to use the same method to map the HVM-loader driver to a protected physical page.

The challenges faced when developing HVM-based security solutions vary according to their monitoring goal: (i) CPU registers can be directly monitored through hypervisor reads and memory access can be monitored through translation faults and IOMMU; (ii) system call tracing requires bridging the semantic gap by employing taint tracking and event analysis; (iii) breakpoints and step-by-step execution require more sophisticated approaches. Breakpoints may be divided as software and hardware breakpoints. Software breakpoints are not transparent since they modify instruction bytes. Hardware breakpoints are limited in number and are shared between host and guest, and so is step-by-step execution. The way tools overcome these limitations is to monitor the system by raising periodic exceptions. One might hook the memory management unit to set a given page as read-only, thus causing a page fault at each instruction execution, or set performance counter registers to their maximum value, in order to raise an overflow exception at each running step.

4.1.1. VMI limits. Despite all the benefits HVM can provide, some limitations are technology-inherent. In this section, we discuss these limitations and how they affect the development of security solutions.

First of all, we should be aware that the transparency claims made by different authors are not totally supported by processor vendors. According to Pearce et al. [Pearce et al. 2013], VM transparency would be achieved by accomplishing the three Popek law requirements: efficiency, resource control and equivalence. Other authors, however, talk about the unfeasibility of such implementations [Garfinkel et al. 2007]. In fact, vendors do not make such claims and often do not consider transparency a major requirement, though a desirable one. This way, an analysis system will be as transparent as the vendor wants it to be.

Second, HVM-support improvements may affect the HVM effectiveness and the way different security techniques are applied. [Lengyel et al. 2014] presents common pitfalls when designing VMI systems and delves into some of these modifications. The first issue refers to TLB splitting, usually sectioned into data (dTLB) and instructions (iTLB). Newer CPUs have a third section called sTLB that caches the evicted/flushed entries from the standard ones, resulting in a considerable performance boost. TLB has been used in the security context for either defense or attack purposes. Grsecurity [Grsecurity 2013], for instance, employed TLB to implement page execution attributes before NX was launched, whereas the Shadow Walker rootkit [Sparks and Butler 2005] achieves its stealthiness through a TLB poison. For this attack, the rootkit remains stealthy by taking advantage of the fact that "a single virtual address can point to distinct pages, according to which TLB is being used". By leveraging this technique, it could, for example, bypass an AV software, since the latter would be unable to scan the malware pages.

Lengyel et al. also point out that an effective defense mechanism against such attacks is to periodically flush the TLB, since this reduces the analysis-time opportunity window. Windows 7 and later versions implement this approach. In addition, sTLB also makes it harder to detect a stealthy rootkit, as the former "can only store one version of the evicted TLB entries". More importantly, VMI may be employed to skip the sTLB. By marking pages as execute-only—an EPT feature—data from iTLB and dTLB will differ and CPU address translation will go through the primed page-tables again, thus restoring stealthy techniques eventually used. Moreover, newer CPUs rely on tagged TLB, a mechanism that labels data in order to manage it, resulting in another performance boost. As the TLB is not flushed anymore, it opens a new opportunity for stealthy rootkits. Another bypassing possibility refers to the EPT mechanism: when a VM-Exit occurs, its violation reason (Read, Write, Execute) is asserted. However, only the start address is specified, thus "an attacker who is able to break the assumption that the violation happened at exactly the pointer location may evade an analysis". As far as we know, Intel is working on such limitation. When building a VMI-based system, one should also care about monitor triggering, since a passive monitor (snapshot-based) may be evaded by a timed execution. [Wang et al. 2015a] presents applications employing this evasion technique, including stealthy file transfer and backdoor enabling.

In summary, despite HVM-based systems having raised the bar against evasive malware, sandboxes may still be detected by advanced threats [Brengel et al. 2016]. A complete discussion of HVM evasion is provided in [Pék et al. 2013]. Even though some evasion tricks have already been mitigated either by introducing new hardware or by leveraging evasion-aware programming guides, the cited work shows how transparent machine vendors are implementing these capabilities.

4.2. System Management Mode

The System Management Mode (SMM) is a CPU operating mode that acts as a mechanism for implementing system control features like power management. The SMM operation abstraction is similar to the one presented for HVM: the system under monitoring executes in ordinary CPU modes (guest-analogous) and the SMM monitor (hypervisor-analogous) in SMM mode. Getting into SMM is triggered by a System Management Interrupt (SMI) and leaving by executing the `RSM` instruction. The RSM exit instruction can only be executed in SMM mode, which protects the code, while the SMI may be triggered in a variety of ways, e.g. by PCI devices, by directly writing to CPU pins, with a periodic timer or by ACPI/APIC interrupts. Some events need to be rerouted in order to trigger an SMI, something that can be done through the chipset or the Interrupt Descriptor Table (IDT). When an SMI is triggered, the whole execution context is saved in the System Management RAM (SMRAM), followed by execution of the corresponding event handler. The SMRAM is protected thanks to its being addressable only in SMM mode: its address range is rerouted to VGA when running in other CPU modes. And as all protected data and code is stored in SMRAM, any development is severely limited to a few KBs.

In SMM mode, the addressing mode provides a direct mapping to physical pages, i.e. no translation is performed. This is quite different from HVM, since SMM code developers or system analysts also need the CR3 registers to bridge the semantic gap of virtual pages. One drawback lies in the fact that memory addressed this way is restricted to 4GB, even in Physical Address Extension (PAE)-enabled systems: the transition to other CPU modes with full access to system memory requires exiting the SMI mode. To overcome this, one may use some kind of subversion or insertion of a callback instruction directly in the program code. The SMM code is initialized by the BIOS, therefore requiring its replacement by an instrumented one using, for instance, Coreboot [CoreBoot 2015] and SeaBios [SeaBIOS 2015].

Building an analyzer in SMM mode is a natural follow-up: this mode is well protected from the "guest" system, which remains completely unaware of its presence and therefore allows composing a strong threat model. Besides, such analyzer would also run instructions directly on the processor with no address translation, but still having memory access. One may think of such system as a more fine-grained, bare-metal analyzer. Its architecture also follows a client-server model, with the SMM code playing a server role and some code on another, network-connected machine playing the client role. As this system runs in a low-abstraction level with no OS support, all communication needs to be implemented from scratch, including network drivers and protocols. Additionally, as data is transmitted through an insecure media, cryptography and error detection/correction also is expected in such communication. Analyzers' initialization may be done using SMI triggering facilities, such as listening to an external serial port or using the Intelligent Platform Management Interface (IPMI). Security features may be implemented by making use of the SMM architecture, according to the desired requirements, such as data watchpoints, I/O monitoring, step-by-step execution and so on. They all work similarly, i.e. by triggering an SMI event for a given action (e.g. breakpoints may be implemented by triggering an overflow event on performance counters).

When building such systems, it is important to reduce the attack surface. Malware that try guessing whether they are running on an SMM-monitored environment must be properly handled. Some malware counteractions may include BIOS overwriting, which may be avoided by employing hardware-assisted Trusted Boot. One must also realize that kernel malware have access to debug and performance counter registers, which may be harmful to an analysis process based on such values. This could be mitigated by periodically triggering SMI to check their values. Another problem is BIOS fingerprinting: as BIOS is rewritten, original hash values and strings are changed. As such, a malware could compare these values to known BIOS-vendor ones to detect the monitoring environment. An effective way to overcome this problem is to perform online BIOS flashing back to the original one, just after the modified BIOS is loaded in memory. It does remain as an open problem for systems that do not allow online BIOS flashing. As for the future, SMM mode will certainly undergo changes, the most significant one being the SMM virtualization (STM), which would deny most SMI requests. A possible solution to this might be rewriting larger parts of BIOS code for earlier event handling, albeit significantly more complex.

4.3. Management Engine and Secure Processor

The Management Engine (ME) is another management mode present in Intel chipsets, originally aimed to support Intel's Active Management Technology (AMT). However, Intel recently started using it for executing system sensitive applications. ME may be seen as an embedded processor with its own timer, RAM/ROM memories and DMA. As this mode cannot address system memory, DMA is used to transfer system data to ME mode. Similarly to the SMM mode, memory is accessed as physical addresses. As ME can externally monitor and control the main processor, it can interfere even in SMM and BIOS codes, besides kernel and userland rings. These capabilities resulted in backdoor suspicions [Wallen 2016]. Despite its capabilities, ME usage for security purposes is, however, still a limited research field with few published articles or available tools as compared to other solutions. Academically, ME was investigated in [Ververis 2010], but a wide range of research still awaits further progress. In addition, environments like ME are not an exclusivity of Intel: a similar solution is present in AMD processors, called Secure Processor [AMD 2016].

4.4. Performance Counters

In this section, we introduce performance counters as lightweight alternatives for security solution implementations. In general, existing counters may be classified in two groups: sampling-based, which counts the number of occurrences of a given event, and trace-based, which logs data on a per-event basis.

4.4.1. Sampling-Based. Intel CPUs include the Precise Event Based Sampling (PEBS) mechanism, a specific-purpose CPU feature that provides detailed information about hardware and software events, such as cache misses, retired instructions, mispredicted branches and others. With PEBS, Model Specific Registers (MSRs) are used to define the monitored events and poll interval/maximum counting value, generating an interrupt if a threshold is reached. In the context of profiling-based solutions, these values are periodically collected to establish a normal behavior used to detect deviations. In the context of tracing-based solutions, such values can be set to their maximum ranges, thus causing overflow at each instruction execution and allowing for step-by-step execution.

4.4.2. Trace-Based. The first real-time, trace-based counter available on Intel processors was the Last Branch Record (LBR). LBR is a set of registers organized as a circular list, which collects source and target addresses of taken branches. LBR allows filtering collected events by branch type (near/far jumps, CALLs, RETs etc.) and by context (kernel, userland). LBR's major drawback lies with the data having to be collected by polling. The Branch Trace Store (BTS) mechanism may be seen as a solution for the latter, since it allows data to be stored on OS pages, raising an interruption when a given threshold is reached and so not losing any data. Understandably, the is concern over BTS generating huge amounts of data. In addition, a common drawback of both LBR and BTS is their system-wide view, with no process filtering. Recently, Intel released Processor Trace (PT), which features the ability to collect not only branches, but general system events too, with interrupt raising when the threshold is reached. Unlike BTS, PT is able to filter processes by PID through using the CR3 register. Moreover, PT employs an efficient encoding mechanism, supplying target branch addresses only for the taken branches, as not taken ones may easily be obtained from the instruction pointer.

4.5. Isolated Rings

As more privileged rings have been used for inspection, privacy-concerned applications had to be moved to where they could not be monitored. As a result, isolated rings/modes were developed in a system-independent manner so that monitoring could not be done by other system facilities. The most notable isolation solution nowadays is Intel's SGX. Isolated rings like SGX, however, are not exclusive for Intel processors: ARM processors have a similar ring named Trust Zone [ARM 2009], used for instance on the recently launched Android Nougat [Crowley 2016]. This section covers details of both solutions.

4.5.1. SGX. Intel Software Guard Extensions (SGX) are hardware features and a set of instructions that allows software to run in an isolated mode—called enclave. This mode is OS-independent, with its own API calls and also encrypted memory pages that are destroyed after use. The SGX underlying crypto systems also allow software verification and attestation, which aims to offer tamper-proof capabilities. SGX's integrity itself is assured by hardware TPM and TXT systems. As SGX is based on a new instruction set, programs should be rewritten to include the instructions that allow for enclave initiation, attestation, execution launch, and destruction. SGX crypto API and SDK use are detailed in [Aumasson and Merino 2016] and [Mandt et al. 2016]; Jain et al. provide a research environment for SGX-based applications that is based on a QEMU extension covering SGX instructions, named OpenSGX [Jain et al. 2016].

4.5.2. TrustZone. The TrustZone environment exhibits similarities to SGX's. As SGX-based systems run applications in and out of enclaves, ARM's TrustZone also has two execution modes: the secure one (trusted) and the normal one (rich). Transitions from the normal to the secure mode are performed by the Secure Mode Call (SMC) instruction. Like SGX, TrustZone also implements memory protections on the MMU: the normal mode cannot access secure zones; conversely, the secure mode can access normal pages. Unlike SGX, TrustZone directly affects peripherals, since Fast Interrupt Requests (FRQ) can only be raised from secure mode.

4.6. Hardware features

Apart from using special processor operating modes and counting on purposely-developed hardware, system monitoring for security purposes may rely on additional hardware features, such as general PCI cards, GPU and transactional memories.

4.6.1. PCI Cards. PCI cards are expansion cards present in most computer systems. Besides performing their specific-purpose actions, such cards are allowed to access the whole system memory through direct memory access (DMA) in a completely stealthy manner, thus being good candidates for security-related tasks. PCI-based DMA access may be implemented through device drivers and/or firmware code. In virtualized systems, the virtualized PCI cards may also be instrumented, as in Xen's dom0. As for limitations, no triggering is available, which suggests this technology is more suitable for snapshot-based data collection. Furthermore, no context information is available from this abstraction level, thus making it harder to bridge the semantic gap.

4.6.2. GPUs. GPUs are PCI devices that, beyond their intended graphic purposes, also enable massive parallel processing. The general programming model of GPU devices is through code offloading, by moving data from main memory to GPU memory and back. Besides using GPU processing capabilities themselves to implement security solutions, analysts can also benefit from GPU memory access capabilities. Since it is a PCI card, it also has DMA access, thus able to access the whole system memory. Moreover, the GPU programming model eases memory transfers as they are routinely done. The existing limitations are the same aforementioned for general PCI cards.

4.6.3. Transactional Memories. Transactional memory is a concurrency control hardware mechanism that allows operations to be executed atomically, applying a concept similar to database transactions. Transactional memory support is present on modern system platforms; we discuss here Intel's TSX solution. TSX is composed by a set of instructions that adds transactional memory support to the x86 architecture. TSX is able to monitor a small region of memory so as to verify whether a transaction may be committed or not. TSX's use requires code rewriting since new instructions (`XACQUIRE`, `XRELEASE`, `XBEGIN`, `XEND`, `XABORT` and `XTEST`) should be used. Due to TSX's monitoring capabilities, it can be used for security purposes by monitoring specific system regions. Its efficiency comes from the fact that transactional memory is asynchronously activated, i.e. it does not need polling. In general, it is also more granular (64 bytes) than trapping the page-fault mechanism, which is 4K-byte granular.

4.7. External Hardware

External monitors are usually implemented as bus snoopers that collect memory data for real-time inspection, either on a snapshot basis or in an event-driven way. They can be deployed using System-On-a-Chip (SoC) in coupled architectures, where a processor monitors the other. As their buses are connected in a single direction, the monitor is tamper-proof and thus not affected by the monitored entity. External monitors, however, need to overcome an important challenge related to the semantic gap: as no context information (registers) is available, introspection should be performed in memory.

5. IMPLEMENTATION CHALLENGES AND TECHNIQUES

In this section, we discuss the general challenges faced by the previously presented technologies and how to overcome them.

5.1. Event triggering

As modern security solutions are event-driven, notification of occurring events is a critical project decision. HVM hypervisors natively provide ways of identifying past events, since each `VM-EXIT` is delivered along the causing exit reason information, such as CR3-register changes, interrupts, memory writes and so on. SMM-based systems must redirect ordinary system interrupts to the BIOS, so the SMM code can be called. can happen on many system Commonly, chipset ports and the APIC controller have their ports rerouted so interrupts are instead delivered to the BIOS. An interesting interrupt handler is the Performance Monitoring Interrupt (PMI). While delivery in the ordinary mode is used by HPCs to handle interrupts at the kernel level, its redirection to SMI delivery mode allows SMM to be called.

5.2. Memory monitoring

Memory monitoring is an essential feature of any monitoring mechanism, since memory holds state/context. The general way of inspecting memory is to rely on page fault traps, used by all of the solutions presented in this article. In this technique, a given page (or all of them) is marked as not present, causing a page fault whenever accessed. The memory management system is instrumented to perform its monitoring intent and supply the data. After data is supplied, the page is marked again as not present, so monitoring may follow on. In addition to be marked as not present, pages may also be marked as having read, write and/or execute permission. This allows for selecting which event will page fault and trigger monitoring. Monitoring goals vary: malware analysis systems, for instance, only log the accessed page whereas debuggers can change memory content before supplying the faulting page. A commonly used technique is called shadow paging and consists of making copies of entire pages or memory regions, before and after each access. Therefore, page comparisons allow for state reconstruction, code unpacking etc. myriad of security tasks. The use of page fault traps or shadow pages in security solutions is not new *per se*, however modern MMUs provide ways of performing such instrumentation by leveraging their hardware capabilities, as seen in HVM-based MMUs.

5.3. Step-by-step

Executing step instructions is an important task when analyzing code, since the analyst can focus on specific code regions. In general, all of the solutions presented here make use of either one of the following techniques, instruction page fault or HPC interrupts. The first approach consists in applying the aforementioned page fault trap technique to instruction pages, by having each executed instruction trigger a page fault and thus allowing step-by-step execution. The second approach consists in setting hardware counters to their maximum values, thus raising exception overflows at each executed instruction. Step-by-step execution is accomplished by repeatedly following this approach. It is used by both HVM- and SMM-based solutions. HVM can natively handle interrupts through VM-EXITS; SMM, in turn, needs redirect overflow interrupts to SMI delivery mode.

5.4. Instruction handling

A drawback of instruction stepping by fault and exception handling is that check granularity is limited to entire pages, instead of a single instruction. Additional data sources must be employed to overcome that. Systems whose register access is direct, such as HVM, can read the faulting Instruction Pointer (IP) in order to identify the last executed instruction. This strategy is used by most HVM-based solutions. Other HVM solutions, as well as SMM ones, may rely on performance counters, mainly branch monitor, for instruction identification. When a fault occurs, SMM code can look at the last branch stored in the branch monitor entries and thus identify the executed instruction block.

5.5. Breakpoints

Breakpoints are extensions from instruction stepping that, by their very nature, require previous definition and are therefore limited in number. To achieve real instruction stepping, breakpoint implementations present some challenges. Breakpoints may be classified as hardware- and software-based. A hardware-based breakpoint is a hardware facility that allows stopping execution when it reaches a given address. The stopping addresses are stored in CPU registers ans therefore limited in number. Besides not being so flexible, they are not scalable—given that they shared by host and guest—and also evadable, as their use is detectable by querying the MSR register. Conversely, software breakpoints exhibit far more flexibility and scalability, considering that they are theoretically unlimited. Nonetheless, they do modify original instruction bytes to include a trap flag, thus being detectable by integrity checks.

In order to implement breakpoints, modern monitoring solutions rely on two main techniques: step-and-compare and invisible breakpoints. The first consists of stepping instructions using the previously presented techniques (fault trap and HPC) and comparing the current IP to previously stored breakpoint addresses, incurring in considerable overhead. The second comprises hiding the side effects of software breakpoints. As an example, the `pushf` instruction needs to be intercepted to hide the trap flag, since its side effects could allow the monitored code to detect this flag.

5.6. Fallbacks

In addition to event-driven mechanisms, other types of triggering may be required, such as callbacks and fallbacks. Code callback insertion is present in SMM-based tools: as the SMM mode is limited to address 4GB, it has to rely on alternative ways to monitor systems with larger amounts of memory. Monitoring may be performed, for instance, by inserting code callbacks into high memory addresses, thus triggering SMI code. To do so, code is injected in mapped pages and EIP is modified—in the `State Save Map` (SSM)—to point to the injected code. Once an `RSM` instruction is executed, `EIP` is restored from SSM and normal system execution resumes from the custom code. Another example occurs when the SMM mode is required to monitor a hypervisor. When an SMI is triggered, the system cannot tell whether the hypervisor is running in VMX Root or Non-Root modes. However, to handle VMCS data the CPU must be in root mode. To overcome this challenge, a fallback technique must be employed to "guarantee that the CPU falls back to VMX root operation". The technique works by redirecting a performance counter overflow interrupt caused by the execution of a fallback code injected in the SMI handler, thus recovering control over the monitored code.

5.7. Granularity filtering

Modern monitoring technologies enable data collection at a very fine-grained level, thus generating massive amounts of data. While this kind of technology allows for more powerful analysis, it may just as well turn simple tasks into more complex ones. To handle this issue, all presented applications provide ways of filtering captured data. These filters, called compact modes, may be applied after data capture to show only desired data, or during data capture to store only some desired events. This is achieved by making use of breakpoints and specific page fault traps, thus also presenting a performance speed up as compared to the verbose, original full mode.

5.8. Timing evasion

Despite execution side effects, malware may also evade analysis procedures by identifying time measurement differences. This comes from analysis procedures imposing significant performance penalties, since they execute both the sample's and the extra monitoring instructions. The first attempts to retrieve system code were based on API calls, which were easily defeated by analysis solutions through faking responses via hooks. Currently, an effective way of monitoring elapsed time makes use of the TimeStamp Counter (TSC), a system-wide register that is incremented on a tick basis. As this counter is system-wide, HVM's guest code may be able to identify that the register was incremented a significant number of times while in VM-EXIT, since hypervisor code execution also increments TSC. To prevent that, all presented HVM- and SMM-based solutions fake TSC values before resuming execution. Advanced samples could in theory be able to identify obvious fake values (such as zero or constant offsets), which is usually avoided by randomly incrementing the TSC offset.

5.9. Footprints

The main advantage of the discussed solutions is "transparency" against detection by execution side effects. However, there are other ways to detect those solutions, such as through the monitor code. Malware that run inside a VM may try mapping the hypervisor's physical memory to scan it and possibly detect the hypervisor monitoring code. To avoid that, hypervisors often mark such pages as not present and block mappings from within. Malware may also detect the presence of the driver used to load hypervisor or SMM code. To address this issue, drivers may use rootkit techniques for stealthiness and hypervisors, in turn, can also map driver pages as not present. Moreover, malware samples may try to guess if they are running on an SMM-based system by checking the use of SMI-triggering performance counters and debug registers. As an evasive measure, they may try to preventively disable such mechanisms. A resilient monitoring system should be able to periodically check whether these mechanisms are still active. In SMM systems, this may be done by using a second source of SMIs, such as a periodic timer.

5.10. Fingerprints

Despite transparency regarding side effects, monitoring solutions may also be evaded through fingerprinting—the identification of the environment as being under analysis, given the presence of known strings, drivers, IP addresses etc. Fingerprinting prevention is an open problem and is beyond this survey's scope. However, the discussed tools implement certain levels of randomization (such as periodic changes in strings and serial numbers) to avoid being fingerprinted. Fingerprinting also affects SMM-based solutions: as the original BIOS is replaced with modified code, samples may detect unexpected strings and hash values. To hide such modifications, many systems try to online flash the BIOS with the original code so as to present ordinary values when probed.

6. APPLICATIONS

In this section we discuss how security solutions can be deployed according to the previously presented technology support, as well as their limitations.

6.1. General Application

A general approach for developing security solutions is to directly rely on new mechanisms as they are launched. Take, for instance, the GPU case, used in several contexts for packer detection [Gupta et al. 2014], IDS implementation [Alshawabkeh et al. 2010; Vasiliadis et al. 2011], security log processing [Bellekens et al. 2014], cryptography [Vasiliadis et al. 2014], AV parallelism [Vasiliadis and Ioannidis 2010] and code polymorphism [Vasiliadis et al. 2015].

Another class of applications that directly benefit from an upcoming technology is the privacy-preserving one, which can be executed within an isolated enclave. The protected chat [Hoekstra et al. 2013] is an example application that takes advantage of Intel's SGX capability: images are processed inside the enclave and therefore protected against external capture. ARM's TrustZone [Yalew et al. 2017], too, benefits from tamper-proofing capabilities provided by isolated enclaves to prevent being it from being disabled.

6.2. Fuzzing, Bug Detection and Crash analysis

Code analysis is a very important security task and can be performed at distinct life cycle stages—pre-deployment, deployed (in production), and post-crash. When analyzing code before deployment, one is mostly focused on checking the program's correctness by validating its inputs and behavior. This is mostly done through program fuzzing [Felderer et al. 2016]. This technique randomizes program inputs in order to exercise all paths [Tsankov et al. 2013; Li et al. 2017], thus branches are important decision points. Although branch monitoring can be performed using plenty of technologies, including VMM and SMM, they are usually costly. Branch monitors, in turn, are more suitable candidates for this task. As an example, Paleari [Paleari 2015] presents the FuzzTrace tool, which leverages the BTS mechanism for achieving binary coverage in an efficient way. He demonstrates that hardware-assisted solutions take only a quarter of the time that PIN solutions do to perform the same task.

When analyzing deployed code, one is mostly focused on bug detection, since program functionality validation is supposed to have been done in previous development steps [Felderer et al. 2016]. Once a bug is discovered, the program is exercised with the buggy input in order to pinpoint the bug source—i.e. the code region which makes the program reach the identified undesired path. The process of following an input flow is named taint tracking. Similarly to the fuzzing case, taint tracking can be implemented by leveraging many technologies, such as emulators [Ho et al. 2006] and even compiler support [Backes et al. 2015]. Branch monitoring, however, is the most suitable choice, since it is a lightweight, specialized monitor for COTS binaries. As an example, [Arulraj et al. 2014] demonstrates how the information provided by the LBR mechanism can be used to perform root cause analysis on buggy programs.

Finally, one can also analyze a program execution after crashing. In this case, more than just showing a buggy behavior, the program was also unable to recover from an unintended state. Crash dumps are system information collected during fault executions which may help bug discovery. Even though there are many existing solutions for crash analysis, recent research have demonstrated that enriching crash dumps increases bug identification success rates. As in the previous cases, branch information is well suited, since branches are directly responsible for taken paths. As an example, [Xu et al. 2017] presents a system which enriches crash dumps with branch data from Processor Tracer. The additional data allows for backwards taint analysis, so simplifying bug cause analysis. Its efficacy was demonstrated through discovery of more than 30 real bugs.

6.3. Malware Analysis

Given the transparency property, the presented environments are suited for malware analysis in a stealthy way. A deeper understanding of HVM for malware analysis uses was provided by [Dinaburg et al. 2008b], establishing formal foundations to attain transparency, from requirements to ways of fulfilling these requirements. The developed tool, Ether, is implemented as a Xen patch and runs Windows XP guests. Its instruction monitoring is done through software breakpoints and step-by-step execution, thus requiring the `PUSHF` instruction behavior to be modified. Memory is monitored through shadow pages. Ether demonstrates its efficacy on stealthily tracing evasive malware samples and performing code unpack. Despite its results, Ether is not a perfectly transparent tool, since some ways of fingerprinting it are known [Pék et al. 2011]. However, most of them are overcome by applying patches or by new VM extensions of modern processors. Moreover, Ether was a sound step towards being ahead of evasion tricks of its time.

CXPInspector [Willems et al. 2012a] is a second step towards HVM-based malware analyzers. It leverages Intel VT-x support on KVM to perform malware analysis on 64-bit Windows 7. Analysis challenges faced on 64-bit Windows Kernel include handling Address Space Layout Randomization (ASLR) and overcoming Kernel Patch Protections hook limitations, for which VMI is an alternative. In addition to malware analysis capabilities, CXPInspector is also able to perform application/system profiling by measuring the execution time spent on each memory page. CXPInspector also presents a more fine-grained concept for memory handling, named Currently eXecutable Pages (CXP), which allows for multiple scopes and granularities of the analysis. The three CXP granularities are: one memory region, a set of memory regions or one single memory page. By capturing transitions and flows among such CXPs, the system can trace events. In practice, it is a way of implementing memory traps based on EPT or NPT facilities. CXP provides case studies of the Purple Haze 64-bit rootkit analysis and a profile of the Apache web server and its modules.

As an evolution of the idea of analyzing malware on HVM, authors started to care about developing an *ad-hoc* malware analyzer, VMM. The motivation was reducing the Trusted Code Base (TCB), i.e. the code that should be trusted *a priori*. General-purpose VMM implements many more features than the one required for malware analysis, such as virtual devices and plenty of drivers. As is known, the larger the numbers of code lines, the more bugs there might be and the more opportunities for malware evasion.

[Nguyen et al. 2009] presented MAVMM, a lightweight, malware analysis VM hypervisor. The TCB is reduced to 4K lines of code on MAVMM, a welcome feature when compared to the millions of lines on well-known VMMs like Xen and VMWare. MAVMM is implemented using AMD-v instructions and runs an Ubuntu Linux as guest system. The hypervisor is loaded at boot time, in contrast to the late launch approach of Ether and others. Memory is protected using nested page technology. The tool is able to extract different features from the system—such as instructions, syscalls and memory accesses—by leveraging single-step execution and handling VM-Exits. Tracing granularity is filtered in two operating modes: Compact and Full. Authors pointed the tool is also transparent since HVM is employed. While this solution is less prone to subversion by presenting a smaller TCB, it is also more prone to fingerprint-based evasion, given that it does not implement many usual virtual devices and so is subject to environment detection.

Beyond just building analysis tools themselves, some authors have employed these tools for analysis improvements. Quist et al. [Quist et al. 2011], for instance, proposed using a modified version of the Ether HVM to improve AV detection accuracy. The work adds to Ether features for "deobfuscation: section and header rebuilding as well as automated kernel virtual address descriptor import rebuilding". With these repair mechanisms, AV showed detection rate improvements as high as 45%.

Notwithstanding its transparency, HVM is an expensive approach. since its implement is harder than in-guest monitors and imposes a greater performance penalty. Conversely, emulators and DBTs, despite being easy to instrument, are not transparent. Therefore, a solution that capitalizes on both would be desirable. Aiming to bridge this gap, V2e [Yan et al. 2012] presents a combined approach of capturing data on an HVM and precisely replaying it on an emulator, which is then able to implement analysis techniques in a more flexible way. It is implemented on an HVM-KVM environment and replayed on TEMU, from both Linux and Windows XP guests.

The most challenging task for this implementation is to achieve a balance between captured data and replay feasibility (considering speed, precision and costs). To that extent, authors have introduced a formal definition of how replaying should look like. For the capture function, the HVM system implementation is based on EPT/TDP/NPT for partitioning the memory space on mutually exclusive recorder and recorded pages. Besides, both TSC and DMA accesses are recorded in order to allow deterministic execution replay.

Additionally, the replayer function featuring a conventional emulator would have a series of problems as compared to a real CPU. It uses block translation (a paradigm absent on real CPUs), lazy flag calculation, translated code reuse and TSC redirection for host values. V2e solves these by disabling lazy flag calculations and replacing unsupported instructions with NOPs. Moreover, as the same page table mechanism used for capturing is required for replaying, a software-emulated one, called physical page container, was developed. Case studies cover `adore-ng` and 12 other real-world malware samples.

A similar approach is implemented by [Kang et al. 2009], replaying Ether instructions on TEMU. This approach allows not only for executing evasive malware but also for determining the points where behavior differs between reference and emulated platforms. By analyzing divergence causes, the tool performs a dynamic state modification (DSM) that attributes "new values to specified execution state components, such as registers, memory and so on, which represent a transient alteration to values during the samples' execution". They use this technique not for removing anti-emulation checks, rather to ensure the sample will run even with distinct inputs, which enables a myriad of security analysis.

Apart from HVM, SMM is also a good candidate for a malware tracing tool. Although solutions able to trace binaries in some way had been built on top of it, such as debuggers, no specific, SMM-based malware tracing solution was developed. The same reasoning applies to ME/AMT, which was demonstrated capable of inspecting systems, but no malware tracing solution is available. Leveraging ME/AMT for malware analysis is a development gap to be explored by researchers.

As an alternative to HVM, stealthier malware tracing may also be performed using performance counters. Their main advantage is reduced overhead, since handling interrupts at kernel level, for instance, is less costly than at hypervisor or BIOS level. In addition, development efforts are also reduced, since no hypervisor/BIOS code is required. Unfortunately, single-step execution cannot be done since the available mechanisms are branch-granular.

By making use of the LBR mechanism, [Willems et al. 2012b] developed `BranchTrace`, a branch monitor able to detect accidental or incorrect behavior of dynamic analysis in an emulated environment. The approach is motivated due to newly developed `delusion-attacks` that are able to detect CPU emulators. They leverage different instruction execution side effects between an emulated and a real machine. The authors remark that the deviating behavior may be fixed in the emulators, but this would require special handling for a variety of instructions. Therefore, branch counters are shown suitable for monitoring bare-metal-based systems.

In BranchTrace, monitoring is triggered at each taken branch, including conditional and unconditional jumps, calls, interrupts and exceptions. The supplied data is the addresses of the source and target branch instructions. The authors claim it is still possible to reconstruct whole contexts from such information. They also suggest extending the information by using Windows debug symbols and disassembling the nearest instructions. For a practical evaluation of this approach, they extended a tool called CWXDetector that is capable of detecting exploitation attempts and extracting shellcode used during exploitations. It resulted in significant information gain about the path that led to such exploitations, as demonstrated on a set of 4,869 malicious PDF documents.

A broad formalization on branch-monitoring-based malware analysis was presented by Botacin et al. [Botacin et al. 2018], which discusses possible threat models—restricted to userland when a kernel driver is used to handle interrupts—and implementation issues. The work presents analysis results considering real samples and demonstrates how anti-analysis tricks and deviating behavior can be identified.

All of the aforementioned mechanisms can also be applied for malware analysis in ARM architectures. The work by [Ning and Zhang 2017] presents a solution for malware analysis which relies on performance counters for data acquisition and the isolated TrustZone for anti-tampering data analysis. Other solutions, such as HVM, may also be employed. In this sense, the Xen on ARM project [Project 2017] is the best candidate for framework implementations.

6.4. Program Debugging

Complete debuggers can also be implemented using HVM and SMM tools. Regarding HVM, [Fattori et al. 2010] presents a complete HVM framework which allows for tools to be built on top of it. Its working mechanisms are very similar to the Ether tool, such as late launch load, shadow memory monitoring and trap flag hiding. The framework is implemented using Intel's VT-x and a client-server architecture, where low-level server information is translated into high-level semantics using introspection procedures and delivered through a well-defined API. In the same article, authors presented HyperDBG, a kernel debugger built on top of that framework. It boasts the same widespread functions of kernel debuggers, such as breakpoints, register inspection and tracebacks, supports guest write access and includes a hypervisor-based graphical user interface and hotkey support. Due to its self-contained implementation, HyperDBG is able to debug any kernel component, including components used in its GUI, such as the keyboard. HyperDBG is a sound step towards transitioning HVM-based security applications from tracing to debugging.

SPIDER [Deng et al. 2013], a KVM implementation of the concept of hypervisor-assisted invisible breakpoints for Windows XP and Ubuntu, is a distinct approach for debugger implementation. It tries to match the flexibility of software breakpoints with the facilities of a hypervisor MMU, allowing for proper handling of the needed instruction changes and control-flow-deviation side effects. This is achieved by splitting the data and code views, which causes a lower overhead than trapping each instruction, like Ether does. Code splitting is performed by setting the same virtual pages to two different physical pages, each one having mutually-exclusive Read/Execution attributes. The data splitting decision is supported by the TLB separation on iTLB and dTLB in the x86 architecture, which reduces the number of EPT violations. SPIDER monitors the system using the monitor trap flag (MTF), which causes a VM-Exit at each instruction. The SPIDER breakpoint handler checks if the current instruction is an ordinary (guest-set) or an invisible breakpoint, acting as a pass-through or a breakpoint hider, respectively. In the latter case, it will "clear the breakpoint and restore the first byte of the instruction that had been replaced". Whenever a write occurs, it happens in a data page, meaning that self-modifying code is forcibly not allowed. This could lead to incorrect execution or even to malware evasion. In order to handle this situation, SPIDER synchronizes data and code pages on writes. Case studies show how SPIDER improves the tamper-resistance of the BEEP [Lee et al. 2013] *attack provenance system*. Also, how SPIDER could be used to monitor instant messaging programs by inspecting them before messages are encrypted.

Regarding SMM-based debuggers, [Zhang et al. 2015] presented MALT, a complete debugging framework based on SMM which implements basic debugging facilities—such as breakpoints, CPU register access and memory examination. It is implemented on a client-server architecture, offering possibilities of communication by using a GDB-like protocol or through its user-friendly interface. Like other SMM-based solutions, MALT bridges the semantic gap using the CR3 register and triggering SMI by rerouting events and overflowing performance counters for instruction stepping. Its protection is achieved by online BIOS flashing and periodically checking MSRs values. MALT is able to provide four different levels of step-by-step debugging: instruction-level, branch-level, far control transfer level and near return transfer level. These modes are shown in the case studies, where complete Windows and Linux kernel crashes are analyzed.

6.5. Forensics

HVM and SMM also have advantages in comparison to ordinary forensic solutions, given their transparency and system-wide views. In particular, HVM is a good candidate for the development of forensic solutions given its late launch capabilities.

HyperSleuth [Martignoni et al. 2010] is an HVM-based solution which benefits from the late launch and unload capabilities to implement a complete forensic framework, for both online and offline inspection. As for analysis tools, the authors implemented a memory dumper—in a dump-on-write way—and a syscall tracer—performed like Ether's by trapping memory pages. Since the system is aimed to be loaded on compromised systems, it has to check whether the procedure was correctly done, given that a malware running on the target system could subvert the procedure. This is done through a sequence of challenges and responses, called "lie detector". It is composed of two data collection components: an in-guest, ordinary process and a hypervisor introspection mechanism. After collection, both results are compared on the trusted host. Since the target system changes at runtime, this verification is performed repeatedly at variable intervals to avoid time measurement attacks. In addition, the framework also cannot trust the OS network software, so it implements its own network driver in order to transmit captured data.

SMM may also be leveraged for forensic purposes. Besides presenting the same capabilities of HVM, it has a deeper system viewer, which allows dumping even VMM memory. As a drawback, SMM does not have late launch capabilities. To overcome this, a modified BIOS is preloaded onto the system and SMM mode monitoring is enabled on demand through hot keys and/or external interrupts. A notable SMM-based forensic tool is SMMDumper [Reina et al. 2012]. It is able to acquire volatile memory contents on running systems and therefore help with digital forensic analysis and incident response. SMMDumper is composed of two components: a trigger, generated by modifying the I/O APIC Redirection Table that is responsible to handle SMI and an external host, which receives collected data. Data transmission is supported by a system-plugged cryptographic device used for signing data to assure data integrity. SMMDumper activation is done by pressing a pre-defined key sequence. In order to implement such functionality, the authors have implemented an SMM-based keylogger. The SMM ISR is responsible for extracting the scancode from the keyboard controller buffer by reading from the I/O port. It also overcomes the SMM-imposed 4GB barrier—to take whole-memory snapshots—by inserting a callback into the monitored code. SMMDumper's evaluation shows it is practical for 6GB dumps.

A slightly different solution is presented by [Wang et al. 2011], which leverages PCI DMA access to complement the SMM mode: it allows whole-system memory analysis and bridging the memory/CPU context gap. Analysis may be performed either online or offline, with the CPU state transmitted from SMM to an external client through a GDB-compatible protocol. Consistency is guaranteed because during SMM the OS enters and remains in suspended state. SMI is triggered through IPMI and memory capture is performed on a dump-on-write way. A limitation of this approach is that PCI card access to physical RAM memory may be blocked on both Intel and AMD platforms by using the newly added MMU features, such as Device Exclusion Vector (DEV).

6.6. Security Policy Enforcement

Empowered by HVM system-view capabilities, some solutions were designed to enforce security policies, particularly badly-formulated I/O policies that are known targets for information leakage attacks. To this end, [Shinagawa et al. 2009] proposed BitVisor, a single para-passthrough[1] VM that enforces I/O security policies. It consists of a specific-purpose hypervisor that implements only essential I/O drivers, reducing the TCB. Bitvisor works on a single VM guest, as authors claim desktop users run only one system at once. As such, it has no need for VM-isolation, which also helps to reduce the TCB and the overhead in general. The para-passthrough approach requires intercepting only essential communications, such as those required for protecting the hypervisor and to enforce the policy itself.

To correctly enforce I/O policies, Bitvisor has to handle 3 different types of I/O routines: programmed I/O (PIO), memory mapped I/O (MMIO) and DMA. PIO are the IN and OUT instructions, handled by the VT-x port bitmap. which allows for intercepting specified ports. It has also to intercept PCI PIO in order to handle port remapping. MMIO device registers are mapped on memory regions, in a way shadow pages fit suitably well. DMA interception is handled by a new technique called shadow DMA. Modern systems use what is called DMA descriptor, a memory region in which DMA controls are mapped. However, monitoring this region is not effective since DMA memory accesses themselves occur in parallel. In order to overcome this situation, Bitvisor instates a shadow DMA descriptor page, mapping the DMA descriptor in hypervisor memory. After copying those blocks to the DMA controller buffer, it follows a Man-In-The-Middle approach against the DMA controller to gain access to all DMA communication. By using these monitoring processes, the authors present a case study of an ATA Host Controller, which enforces automatic storage encryption. SMM can also be used to ensure I/O integrity, which means known ports will not be mapped to other ones to avoid potential malicious actions. Such integrity is relevant since, after compromising an I/O controller, attackers will be able to change memory via DMA or by compromising I/O devices.

Trusted Platform Modules (TPMs) are able to protect firmware and IOMMU integrity at boot time, but not at runtime. The Input/Output Memory Management Unit (IOMMU) tries to protect memory from DMA attacks. However, the root entry table's

[1]A minimal interposition mechanism responsible for I/O filtering

base address and other configuration registers may also be under the attacker's control on specific scenarios. Besides, the National Vulnerabilities Database (NVD) [Nist.gov 2017] shows that many firmware vulnerabilities were discovered since 2010, therefore enlarging the attack surface. As such, authors have proposed I/O Check [Zhang 2013], a solution which employs SMM to check I/O configurations and firmware integrity by enumerating all I/O devices. I/O Check assumes the system is supported by a TPM hardware for its boot and also assures BIOS image integrity. It also assumes that SMRAM is locked in the BIOS after loading. Its verification has as base the premise that the "DMA Remapping ACPI table should never change after booting" and that "the base address of the configuration tables for the DMA remapping unit should be static". Attack detections are notified through audible beeps. It ensures NIC integrity by storing its original hash value in SMRAM and by periodically reading the NIC's memory firmware code, computing the current image's hash value and comparing it with the saved value. Trusted Storage, in general, is an issue for all platforms, including mobile ones. Current solutions [Zhauniarovich et al. 2013], however, do not yet benefit from existing hardware facilities, thus presenting open development opportunities.

6.7. Attack Detection and Prevention

VMM-based system protection techniques have been known for quite some time: Kernel Guard [Rhee et al. 2009] is a framework for handling dynamic kernel rootkits through memory access policies; Lares [Payne et al. 2008] extends Xen dom0 with a new, secure VM TCB providing secure services for an unprotected guest; Osck [Hofmann et al. 2011] defeats kernel rootkits by checking control-flow integrity; NICKLE [Riley et al. 2008] leverages mixed-page techniques to ensure trusted-code execution; Overshadow [Chen et al. 2008b] introduces the concept of multi-shadowing physical pages to protect applications from untrusted kernels. These approaches, despite being theoretically correct, are not fully transparent. This comes either by the need to implement all memory and I/O management by software or by a possible malware evasion through virtualized-system detection. In addition, software implementations present greater overhead when compared to hardware ones. Thus, as soon as the HVM extension was launched, VMM approaches migrated to it.

Secvisor [Seshadri et al. 2007] protects the kernel against injections and 0-days by ensuring that only approved code may run, thanks to memory virtualization. User memory is marked as executable in user mode but not in kernel mode, in a way Secvisor needs intercepting all transitions in order to adjust flags. Secvisor also ensures that switching to kernel mode will occur only by setting the `IP` "to an address within the approved code". Likewise, kernel exits should target only user-mode code, avoiding kernel-flow redirections. In addition, Secvisor virtualizes and intercepts MMU and IOMMU, protecting against DMA and memory writes by using the AMD Device Exclusion Vector (DEV) feature. In order to handle specific actions such as module loading, kernel code modifications are needed, since these actions may require users' approval through a hypervisor call (hypercall). The need for a patch may be considered a limitation by some but allowing users to create their own policies is thought as a justifiable trade-off.

Other solutions rely on SMM-based implementation. Given SMM transparency and its hardware protection, it becomes a suitable environment to malware attack analysis and identification. For this purpose, [Zhang et al. 2013] presented SPECTRE, an SMM tool that allows memory inspection. Instead of relying on code execution approval, SPECTRE implements signature-based attack detection by performing a series of regex-based scans on system memory. It has rules written for heap spray, buffer overflow and rootkit (kernel integrity) detection. Shellcode's NOPs are used as identifiers for memory overwriting. As a drawback, SPECTRE uses system timers to periodically generate SMIs, which may facilitate malware evasion if it acts on such time intervals.

Another potential solution for attack detection is presented by POSTER [Stewin et al. 2011], which suggested using the GPU DMA capabilities in order to detect DMA malware. The approach consists in trying to identify DMA side-effects in registers like timestamp counter (TSC) and HPC. Although a preliminary work, it was a first step towards taking advantage of this possibility. We notice that an approach to detect DMA malware could also explore the HVM system resources, such as IOMMU monitoring, presented in the previous sections. An extension of the GPU DMA monitoring approach was given in the work by [Koromilas et al. 2016], which leveraged these capabilities to perform kernel monitoring on a periodic snapshot basis. Finally, mobile applications can also enforce code execution policies. Existing solutions, however, do not benefit from underlying hardware support: FireDroid [Russello et al. 2013], for instance, implements a syscall policy using the well known `ptrace` support.

Control Flow Integrity. Injection attacks, such as Return-Oriented Programming (ROP), are one of main threats to current systems. Control Flow Integrity policies emerge as ways of mitigating the imposed risks by ensuring execution flow returns only to allowed call sites. In this sense, CFI policies are special cases of the general attack detection and prevention class. Since this kind of detection policy must take place on end-users' machines, lightweight monitoring approaches are required, thus performance counters are the best candidates for solutions development. In particular, branch monitors are well suitable due to their ability to follow taken branches, including those caused by `RET` instructions. Most HPC-based solutions apply a `CALL-RET` policy, which states a given `RET` must be preceded by a `CALL`. Approaches like `CFIMon` [Xia et al. 2012], `KBouncer` [Pappas et al. 2013] and `ROPecker` [Cheng et al. 2014] use the branch record mechanism to enforce strict CFI policies. Other approaches [Pierce et al. 2016] address the ROP problem by using the branch misprediction monitor, since ROP gadgets (their RET targets) are not well distributed in memory and thus cause prediction errors. Another technology that allows CFI implementation is TSX, as presented in [Muench et al. 2016]. The main advantage of this approach is that the malicious transaction is detected—due to violation of the CFI policy—and blocked—in TSX terms, not committed—, keeping the system in a secure state. TSX research is still taking place, but it is easy to imagine security-focused use cases, such as [Birgisson et al. 2008], which proposes a memory introspection mechanism.

Side-effect Detection. An emerging class of security solutions for attack detection and prevention is the one based on side-effect detection, i.e. indirect observations. When an attack happens, the running machine undergoes many architectural events, e.g. branch prediction misses, cache flushes, pipeline stalls and so on. As these events are noticeable, profile-based approaches are successful in identifying their occurrence. For that, a profile (or baseline) of architectural events is generated in a clean system state and then compared to their values while in production. Its implementation benefits a lot from performance counters. One such implementation is presented by [Kompalli and Sarat 2014], in which a Vtune extension was developed to monitor the Branch Prediction Unit (BPU). Initially, a baseline is defined by running benign applications on the system as a training set. Afterwards, a modified version of the `Win32/Renos` malware was evaluated. Results show that "branch prediction miss rates are below threshold for a clean system". However, in infected systems, "BPU produces a high rate of prediction misses". The same approach was extended to runtime memory allocation and usage. An additional extended approach is presented with HPCHunter [Bahador et al. 2014], which uses HPC data to build a support vector machine (SVM)-based event feature selection for real-time malicious program detection. Many authors addressed the latter [Yuan et al. 2011; Demme et al. 2013] and even extended it to the kernel [Wang and Guo 2016]. The biggest advantage of the performance monitoring approach is its lower overhead when compared to other solutions [Malone et al. 2011], whereas their biggest challenge is feature selection [Tang et al. 2014].

6.8. System Integrity

An important management task is to assure system integrity. Apart from preventing attacks (security), ensuring system integrity also contributes to proper working (safety) and avoids execution degradation (performance). Integrity checks implementation can rely on many technologies: performance counters, for instance, can be used like in `Confirm` [Wang et al. 2015b], an approach to validate firmware on embedded systems. Due to HPCs' nature, this approach is profile-based. Another possibility is to rely on TSX, which is efficient performance-wise and can be used to efficiently monitor threads' memory [Muttik et al. 2014]. Nonetheless, TSX is limited to very small memory regions. These are emerging approaches and therefore not yet fully developed. The established way of performing such monitoring is to rely on external hardware. In summary, these approaches work by collecting system data and then comparing it to known/expected values, raising alerts when violations are detected.

The first widely-recognized attempt to implement an external, hardware-based security monitor was `Copilot` [Petroni et al. 2004]. It aims to assure kernel integrity by using a PCI card to collect memory data snapshots and analyze them. As dedicated hardware, it is intrinsically protected against tampering. Its architecture follows the well-known client-server model, where the monitor is responsible for analyzing the received data and identifying threats. As an external solution, this approach has the disadvantage of not getting CPU register values, which limits context comprehension and introspection. Nevertheless, it was a first step towards overcoming virtual address translation on approaches using external hardware, achieved by deriving page information from the Linux's `System.map` file. The developed prototype helps the authors clearly state their approach's main benefit: minimal processing overhead; measurements indicated only 1%.

Besides these advantages, Copilot-like approaches have a significant drawback related to their snapshot characteristic, rendering them susceptible to timing attacks. To overcome this, [Moon et al. 2012] proposed Vigilare, a System-On-a-Chip (SOC) implementation that snoops the memory bus in order to perform real-time analysis for kernel integrity evaluation. Aiming at giving a better understanding of the issues related to transient attacks, the authors implemented two Vigilare versions, each implementing a distinct capture strategy: (i) Snapmon, a snapshot-based, straightforward implementation of Copilot's approach; and (ii) SnoopMon, a snoop-based solution. Experimental results have shown that the snapshot approach, even through the use of a randomized snapshot interval, is susceptible to transient attacks. Its detection rate highly depends on luck, whereas SnoopMon, as a snoop-based solution, is able to detect all attempts. As a snoop-based solution, however, Snoopmon faced a significant challenge: *how to handle lots of data at once?* If Vigilare could not analyze all the bus traffic that snooper provided, the results would be compromised. This way, the tool was designed to have a selective bus traffic filter. which recognizes only meaningful information while truncating unnecessary data. This approach also allowed snooper to filter data on traffic bursts.

Vigilare also proposed two ways of protecting its memory content, be it data or instructions. Firstly, it uses a separate hardware memory, with no guest access; secondly, "it implements a memory region controller which specifically drops all memory operation requests from the host system". The latter may reduce hardware costs in comparison to the former. Despite Vigilare's effectiveness against static kernel code modification, it is not capable of handling dynamic kernel modifications, such as process list changes—a typical rootkit behavior. TO circumvent this, [Lee et al. 2013] proposed Ki-mon, “an event-triggered verification scheme for mutable kernel objects”. Its memory acquisition is performed through a structure called Value Table Management Unit (VTMU) which, besides snooping the bus, is also able to filter its capture and perform DMA access. It also presents callback verification routines that can be instrumented to handle specified events. This is named hardware-assisted whitelisting (HAW) and its registers can be configured to be active in different ways, including a pass-through operation. Callback configuration is performed through a well-defined API. By relying on it, detection rules were developed and tested with real rootkits to check the solution's effectiveness.

As a general summary on hardware approaches, we notice that, although kernel integrity is an essential issue to be addressed, bus monitoring applications should be more deeply explored. A natural scenario seems to be extending such data-invariant checks from kernel to hypervisor integrity monitoring, such as on HyperSentry and HyperCheck solutions, presented below. Finally, there are other approaches that inspect memory traffic by using other hardware features, such as Processor Trace capabilities. Those, however, follow the same previously-presented working principles; Kargos [Moon et al. 2016], for instance, is a high frequency snapshot-based solution.

Hypervisor Integrity. In addition to general system integrity, a security solution for a modern computer stack should also worry about hypervisor integrity, since attacks to these are well known and widely deployed today. [Rutkowska and Wojtczuk 2008], for example, presents an attack to the Xen Hypervisor by redirecting memory reads/writes from the internal guest to the host. [Sharkey 2016], in turn, presents attacks able to trap special instructions under secure hypervisors. This way, protecting hypervisors from attacks and corruption is very relevant for security systems. Given the SMM mode's nature, it is well suited for this purpose, to the way of being employed by a variety of tools, with some of which presented below. One of these is HyperSentry [Azab et al. 2010], a hypervisor integrity checker for cloud environments. Its architecture consists of an agent inside the hypervisor and a client in SMM. The agent transmits to the client chunks of memory data and hash calculations, which are then compared to expected values. Despite being able to access memory, Hypersentry has to overcome the significant challenge of bridging the hypervisor semantic gap from within the SMM mode. Additionally, in order to inspect hypervisors in root mode, a fallback technique must be employed. The study presented a case of monitoring the Xen hypervisor: the authors verified its code integrity using SHA-1, its control flow pointers in the IDT and whether its physical memory guest isolation was functional. Implementing this kind of system, however, still presents many challenges in order to be practical. Hypersentry, for instance, has some limitations: protected registers from Intel's TXT platform were not used; cache was not used in order to prevent cache poisoning attacks; when in SMM mode, interrupts are disabled, which may lead to a crash if it lasts too long; in a multiprocessor scenario, when monitoring an event on a specific core, other cores are frozen to ensure consistency. These are open implementation challenges to be addressed by the research community. Some improvements were presented by HyperCheck [Wang et al. 2010], a hypervisor integrity solution which uses a PCI DMA card to collect memory and SMM collected register data to handle virtual address translation. The system was implemented using two prototypes: the first is an NIC emulation on QEMU and the other is a real PCI NIC. The system also has an analyzer to which data is transferred through the network card. This transfer is protected using a random hash in order to avoid replay attacks, with the key being locked in SMRAM. In order to prevent attacks where a fake device asks for the key, TPM hardware may be used. In addition, a random-interval scan is performed to avoid timing attacks. The study case provided was DMA attacks against the Xen Hypervisor, having both

Linux and Windows XP as guests. However, some kinds of attacks are not detected or prevented by this technique, such as dynamic function pointers or returned-oriented attacks.

6.9. Existing and upcoming threats

Although analysts can benefit from the presented solutions, attackers may also employ the same technology for malicious purposes. For example, given the transparency and system-wide view of HVM systems, exploiting a system using this technique is straightforward. It is hard to pinpoint who first proposed this use for HVM, since most releases happened in underground hacker forums. Undoubtedly, the first famous approach was the BluePill rootkit [Rutkowska 2006]. Bluepill is implemented on Windows Vista using AMD technology and is able to perform late launch; a network backdoor, with no need for NDIS modification, is shown as use case. Other examples appeared, like the HVM Rootkit [Myers and Youndt 2007], which is AMD-based and targets Windows XP machines. This tool takes a multi-core approach, setting up each core for an HVM. Its driver loading routine employs physical page mapping for stealthiness. In fact, in-guest, ordinary kernel rootkits are stealthy enough against casual analysis, but to remain stealthy before a specialized forensic procedure requires HVM-based ones.

In the same way HVM was employed for malicious purposes, SMM has already been targeted. Its attractiveness comes from the same virtues exhibited by HVM: transparency, system-wide instrumentation capabilities and OS-independency. An SMM rootkit also has advantages in concealing its memory footprint, given that SMRAM is hardware-protected, and in surviving reboots and re-installations, since it is BIOS-stored. Probably the first work referring to an SMM rootkit, [Duflot et al. 2007] showed a privilege-escalation attack against x86 OpenBSD. In this attack, the authors bypass secure-level protections by installing their own SMM handlers, allowing unrestricted access to physical memory. Following that, the practical Phrack magazine highlighted some work intended to handle SMM for possibly malicious purposes, as in [BSDaemon et al. 2008] and [Wecherowski 2009].

[Embleton et al. 2008] presents the construction of an SMM keylogger by redirecting the keyboard Interrupt Request (IRQ) on the chipset to SMM using the Advanced Programmable Interrupt Controller (APIC). The pressed keys are logged and transmitted through the network interface. An advantage of this technique over ordinary keyloggers is that no IDT hooking is employed, since one can have an out-of-band access through the chipset APIC redirection. As in previously-presented SMM-based approaches, the network card operation has to be manually implemented, working in a client-server way on the PCI bus and encapsulating data in UDP packets, which are then transmitted when buffers are full. Another SMM keylogger is presented by [Schiffman and Kaplan 2014]. In this implementation, the authors perform an early USB hijack, which allows for interception to occur before the kernel is aware of the event. This is achieved by having the USB Host Controller reroute the interrupts to a USB-PS/2 emulation SMM handler. It constitutes a much stealthier way of hijacking the keyboard events, since keystrokes could be successfully intercepted, replaced and injected. As extensions, authors point the approach could be used to hide and perform Man-In-The-Middle (MITM) attacks against USB devices.

Keyloggers can also be implemented using GPUs, as presented by [Ladakis et al. 2013], in which DMA is remapped to be accessible from the GPU with no hook required. Once the GPU is aware of the keyboard buffer location, it can retrieve data directly from the system's memory pages. Since GPU usage has grown tremendously in the last few years, we consider this kind of threat as a very relevant aspect to be considered in security systems. The GPU keylogger works because it can map any memory address, and this system-view characteristic is what drives most attackers. Therefore, the more privileged operation modes, such as HVM, SMM and DMA-based ones, will certainly be abused by attackers. In this sense, SMM has been criticized for allowing system subversion in many ways, such as attacking the TXT subsystem, which would allow complete system subversion. A mitigation—SMM virtualization—was proposed as a way of sandboxing SMI requests. Authors like Rutkowska, however, claim sandboxing is not enough, since escapes and backdoors are always possible to be implemented [Rutkowska 2015].

Similar criticism also encompasses AMT/ME. A talk on BlackHAT [Tereshkin and Wojtczuk 2009] presented the use of the ME mode to implement a system rootkit. In practice, ME implementation flaws [Hruska 2016] may allow an attacker to take control of the victim's machine at a very deep level. Intel's response was to make available its CHIPSEC tool [CHIPSEC 2016], intended to assure correct configuration of hardware parameters in order to make the system more secure [CHIPSEC 2014]. Such releases, however, are just steps of a continuous arms-race. Right after such discussion took place, the first malware which leverages AMT/ME for stealthy data exfiltration was discovered [Khandelwal 2017].

Since the criticism against these modes relies on the fact they are able to monitor all previously existing rings, new solutions like isolated rings, e.g. SGX, were proposed that cannot be monitored. This characteristic, however, enables other potential threats, such as malware samples which cannot be analyzed. [Davenport and Ford 2014] presents the idea of malware attestation, in which the attacker can attest its malicious payload was not tampered with. In addition, Van Prooijen's work [van Prooijen 2016] illustrates such attestation using SGX. It also points at the hardness of reverse engineering SGX running code. This scenario is bound to bring about new research in coming years since it is currently an open question.

Currently, information retrieval from isolated enclaves is possible only through side-channel attacks. [Schwarz et al. 2017] presented a malicious sample able to retrieve RSA keys from co-located enclaves by monitoring cache access patterns. An in-development alternative is to rely on the branch counter, since SGX shares the same CPU as ordinary code. [Lee et al. 2017] presents the usage of Processor Trace in order to infer program behavior inside SGX enclaves.

7. DEVELOPMENT GAPS AND RESEARCH OPPORTUNITIES

In this section, we revisit development gaps presented in the paper and pinpoint existing research opportunities.

HVM has a great potential for security solutions development as well as many open issues to be addressed. Among them, the well-known issue of transparency claims that are not supported by the vendors, but new ones are showing up. An emerging one is nested virtualization support: questions such as "*How do we support a VM inside another one?*" and "*How do we bridge such coupled semantic gap?*" are still unanswered.

We are also aware of a trend on moving to VM: Qubes [Rutkowska 2010] is a modified Linux OS that isolates each application on a VM; Windows 10 has implemented the concept of Virtual Secure Machines (VSMs), "loading a microkernel with its own drivers, called Secure Kernel Mode (SKM) environment" [Ionescu 2015]; the Edge browser also moved to a micro-VM environment [Dent 2016]; Samsung, in turn, implemented a hypervisor-based approach for kernel protection [Samsung 2017]. If this trend consolidates as a `de facto` standard, we will have another turn, since just detecting VM environments will not be enough for malware authors: they will have to detect the monitoring process itself, which is much harder.

SMM was employed by many security applications, but no specific-purpose, malware tracer was presented. This the first existing development gap we can point out.

The use of SMM to other applications is depending on implementation issues, since BIOS rewriting is costly. As new ways of doing so emerge, SMM use will become more popular. An application class which will benefit from this expansion will be hypervisor integrity systems. Current solutions are mostly hash-based verifications. As an open research question, bridging the coupled semantic gap would allow for SMM to inspect not only the host system but also understand the hypervisor guest semantics.

ME/AMT was presented as a technology able to monitor and control the main processor from outside. Despite all the claims, no security-focused, specific-purposed tool was proposed to benefit from it. Therefore, researchers can leverage ME/AMT for all security application classes aforementioned. In particular, malware analyzers and debuggers are application that might benefit from its transparency.

SGX has been claimed as the solution for privacy issues. Nevertheless, it allows new threats by malware, since enclaves cannot be monitored. Therefore, proposing solutions for enclave monitoring is an open research problem. In particular, as it cannot be done directly, side channel approaches are good candidates.

Performance Counters are another technology that may benefit from side effect approaches. Profile-deviation based on HPCs were already proposed, but recent advances in machine learning may boost new deviation detection algorithms.

TSX presents great potential to be explored by security solutions, since it does not impose significant overhead and is able to not commit data when a given policy is violated. Currently, it is under-explored, being limited to flow monitoring, but we foresee its application into a variety of checkpoint-based approaches. As its major limitation, TSX is able to monitor only a few KB of memory, thus efficient policies should be implemented in order to detect threats based on a small trace of collected data.

DMA monitoring is a powerful approach for system-wide monitoring. The main drawback is to bridge the semantic gap without having context registers. Although many advances have been recently presented, introspection, as a whole, is still an open problem. Also, as DMA allows monitoring for good purposes, it also allows malicious entities to sniff communication. Therefore, ways of monitoring and blocking DMA requests should be developed. As a challenge, data burst may turn snoop-based memory monitoring unfeasible.

External hardware devices is another class of under-explored technology for security solutions implementation. Their tamper-proof characteristics make them strong candidates for almost any security application, but they are mostly applied to isolated execution scenarios, such as in Google's authentication module [Xin 2017]. Therefore, their development for system monitoring tasks presents many research opportunities. As the major challenge to be overcome, researchers have to turn passive solutions into active devices, so that detected threats can be blocked.

Threats will also be developed on top of the presented technologies. While existing threats are mostly of the keylogger and rootkit kinds, these can be extended to general threat classes. The study of such threats, known as offensive security, provides ways of better understanding their workings and then to develop effective countermeasures. Threats focusing on system monitoring are outside this work's scope, though they also pose significant challenges for researchers. While the hereby cited approaches solve most of the timing problems when transparently analyzing a system, external approaches still remain effective, such as NTP time measurements over encrypted connections.

8. CONCLUSION

We presented a complete overview of early-launched monitoring tools for modern systems. We deeply looked into HVM and its applications, as well as SMM, HPC and isolated rings, including threats and defense mechanisms—a comparison of which is summarized in the attached appendix. We do believe that this work will help researchers improve knowledge in the field, since there are still many unsolved issues.

REFERENCES

Malak Alshawabkeh, Byunghyun Jang, and David Kaeli. 2010. Accelerating the Local Outlier Factor Algorithm on a GPU for Intrusion Detection System. In *Proc. 3rd Work. on GP-GPUs*. ACM.

AMD. 2013. *AMD64 Architecture Programmer's Manual Volume 2*. AMD.

AMD. 2016. AMD Secure Processor (Built-in technology). https://tinyurl.com/yaq2rhmv. (2016).

ARM. 2009. *ARM Sec. Technology - Building a Secure System using TrustZone Technology*. ARM.

Joy Arulraj, Guoliang Jin, and Shan Lu. 2014. Leveraging the Short-term Memory of Hardware to Diagnose Production-run Software Failures. *SIGARCH Comput. Archit. News* 42, 1 (Feb. 2014).

Warwick Ashford. 2010. Malware growth reaches record rate. https://tinyurl.com/y8mxxo3e. (2010).

JP Aumasson and Luis Merino. 2016. SGX Secure Enclaves in Practice: Sec. and Crypto Review. (2016).

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 *Proc. 17th ACM Conf. on Comp. and Comm. Sec. (CCS '10)*. ACM.

Michael Backes, Oliver Schranz, and Philipp von Styp-Rekowsky. 2015. POSTER: Towards Compiler-Assisted Taint Tracking on the Android Runtime (ART). In *Proc. of the 22Nd ACM SIGSAC Conf. on Comp. and Comm. Sec. (CCS '15)*. ACM.

M.B. Bahador, M. Abadi, and A. Tajoddin. 2014. HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *2014 4th Intl. Conf. on Comp. and Knowledge Engineering (ICCKE)*.

Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symp., San Diego, USA*. EURECOM, Article -.

U. Bayer, C. Kruegel, and E. Kirda. 2006. TTAnalyze: A tool for analyzing malware. In *15th European Inst. for Comp. Antivirus Research (EICAR 2006) Annual Conf.* EICAR.

Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proc. USENIX Annual Technical Conf. (ATC '05)*. USENIX Association.

Xavier J. A. Bellekens, Christos Tachtatzis, Robert C. Atkinson, Craig Renfrew, and Tony Kirkham. 2014. GLoP: Enabling Massively Parallel Incident Response Through GPU Log Processing. In *Proc. 7th Intl. Conf. on Sec. of Information and Net. (SIN '14)*. ACM.

Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. 2008. Enforcing Authorization Policies Using Transactional Memory Introspection. In *Proc. 15th ACM Conf. on Comp. and Comm. Sec. (CCS '08)*. ACM.

Marcus Botacin, Paulo Lício De Geus, and André Grégio. 2018. Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging. *ACM Trans. Priv. Secur.* 21, 1, Article 4 (Jan. 2018), 30 pages. DOI:http://dx.doi.org/10.1145/3152162

Michael Brengel, Michael Backes, and Christian Rossow. 2016. Detecting Hardware-Assisted Virtualization. In *Proc. 13th Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721 (DIMVA 2016)*. Springer-Verlag New York, Inc.

BSDaemon, coideloco, and D0nad0n. 2008. System Management Mode Hack - Using SMM for "Other Purposes". https://tinyurl.com/jxeao4u. (2008).

Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. 2008a. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE Intl. Conf. on Depend. Syst. and Net. With FTCS and DCC (DSN)*. IEEE.

Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008b. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Syst. *SIGPLAN Not.* 43, 3 (March 2008).

Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. 2014. Ropecker: A generic and practical approach for defending against ROP attacks. In *Symp. on Net. and Dist. System Sec. (NDSS)*. Internet Society.

CHIPSEC. 2014. CHIPSEC Platform Sec. Assessment Framework. https://tinyurl.com/nwxzudm. (2014).

CHIPSEC. 2016. CHIPSEC. https://github.com/chipsec/chipsec. (2016).

CoreBoot. 2015. CoreBoot. http://www.coreboot.org/. (2015).

Paul Crowley. 2016. Pixel Sec.: Better, Faster, Stronger. https://tinyurl.com/y88book8. (2016).

Shaun Davenport and Richard Ford. 2014. SGX: the good, the bad and the downright ugly. https://tinyurl.com/z8jlk3s. (2014).

John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the Feasibility of Online Malware Detection with Performance Counters. *SIGARCH Comput. Archit. News* 41, 3 (June 2013).

Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization. In *Proc. 29th Annual Comp. Sec. Applications Conf. (ACSAC '13)*. ACM.

Steve Dent. 2016. Microsoft's Edge browser stays secure by acting as a virtual PC. https://tinyurl.com/z8j3krc. (2016).

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008a. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. of the 15th ACM Conf. on Comp. and Comm. Sec. (CCS '08)*. ACM.

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008b. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. 15th ACM Conf. on Comp. and Comm. Sec. (CCS '08)*. ACM.

Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proc. 2011 IEEE Symp. on Sec. and Priv. (SP '11)*. IEEE Comp. Society.

L. Duflot, D. Etiemble, and O. Grumelard. 2007. Using CPU System Management Mode to Circumvent Operating System Sec. Functions. https://tinyurl.com/y7mlduy9. (2007).

DynamoRIO. 2001. Dynamic Instrumentation Tool Platform. https://tinyurl.com/ybenfvw9. (2001).

Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2 (March 2008).

Shawn Embleton, Sherri Sparks, and Cliff Zou. 2008. SMM Rootkits: A New Breed of OS Independent Malware. In *Proc. 4th Intl. Conf. on Sec. and Priv. in Communication Netowrks (SecureComm '08)*. ACM.

Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. 2010. Dynamic and Transparent Analysis of Commodity Production Syst.. In *Proc. IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '10)*. ACM.

Michael Felderer, Matthias Büchler, Martin Johns, Achim D. Brucker, Ruth Breu, and Alexander Pretschner. 2016. Sec. Testing: A Survey. (2016).

David Fitzpatrick and Drew Griffin. 2016. Cyber-extortion losses skyrocket, says FBI. https://tinyurl.com/y8ym4q46. (2016).

Yangchun Fu and Zhiqiang Lin. 2013. Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. *ACM Trans. Inf. Syst. Secur.* 16, 2 (Sept. 2013).

Yuxin Gao, Zexin Lu, and Yuqing Luo. 2014. Survey on malware anti-analysis. In *Intelligent Control and Information Processing (ICICIP), 2014 Fifth Inter. Conf. on*.

Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proc. 11th USENIX Work. on Hot Topics in Operating Syst. (HOTOS'07)*. USENIX Association.

Grsecurity. 2013. Grsecurity. https://grsecurity.net/. (2013).

Neha Gupta, Smita Naval, Vijay Laxmi, M.S. Gaur, and Muttukrishnan Rajarajan. 2014. P-SPADE: GPU accelerated malware packer detection. In *Priv., Sec. and Trust (PST), 2014 Annual Intl. Conf. on*. IEEE.

Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical Taint-based Protection Using Demand Emulation. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conf. on Comp. Syst. 2006 (EuroSys '06)*. ACM.

Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proc. 2nd Intl. Work. on Hardware and Architectural Support for Sec. and Priv. (HASP '13)*. ACM.

Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. 2011. Ensuring Operating System Kernel Integrity with OSck. In *Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Syst. (ASPLOS XVI)*. ACM.

Joel Hruska. 2016. Report claims Intel CPUs contain enormous security flaw. https://tinyurl.com/zdlbbvq. (2016).

Intel. 2013. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel.

Intel. 2015. Pin - A Dynamic Binary Instrumentation Tool. https://tinyurl.com/m685m25. (2015).

Alex Ionescu. 2015. Battle of the SKM and IUM: How Windows 10 Rewrites OS Architecture. https://tinyurl.com/na375ur. (2015).

ISECLAB. 2010. Anubis - Malware Analysis for Unknown Binaries. https://anubis.iseclab.org/. (2010).

P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. 2016. OpenSGX: An Open Platform for SGX Research. In *Proc. 2016 Annual Network and Distributed System Sec. Symp.* Internet Society.

Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. 2009. Emulating Emulation-resistant Malware. In *Proc. 1st ACM Work. on Virtual Machine Sec. (VMSec '09)*. ACM.

Swati Khandelwal. 2017. First-Ever Data Stealing Malware Found Using Intel AMT Tool to Bypass Firewall. https://tinyurl.com/y7e7kg8v. (2017).

Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. BareBox: Efficient Malware Analysis on Bare-metal. In *Proc. 27th Annual Comp. Sec. Applications Conf. (ACSAC '11)*. ACM.

Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proc. 23rd USENIX Conf. on Sec. Symp. (SEC'14)*. USENIX Association.

Kompalli and Sarat. 2014. Using Existing Hardware Services for Malware Detection. In *Proc. 2014 IEEE Sec. and Priv. Work.s (SPW'14)*. IEEE Comp. Society.

Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. 2016. *GRIM: Leveraging GPUs for Kernel Integrity Monitoring*. Springer Inter. Publishing.

Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2013. You Can Type, but You Can't Hide: A Stealthy GPU-based Keylogger. https://tinyurl.com/cbzp42n. (2013).

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 *22nd USENIX Sec. Symposium*. USENIX.

Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. High Accuracy Attack Provenance via Binary-based Execution Partition. In *20th Annual Network and Distributed System Sec. Symp., NDSS 2013, San Diego, California, USA, February 24-27, 2013*. Internet Society.

Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Sec. Symposium (USENIX Sec. 17)*. USENIX Association.

Tamas Lengyel, Thomas Kittel, George Webster, and Jacob Torrey. 2014. Pitfalls of virtual machine introspection on modern hardware. In *1st Work. on Malware Memory Forensics (MMF)*. ACM.

Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *Proc. of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM.

LibVMI. 2015. Introduction to libVMI. https://tinyurl.com/y8d4xbq9. (2015).

Corey Malone, Mohamed Zahran, and Ramesh Karri. 2011. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. In *Proc. 6th ACM Work. on Scalable Trusted Computing (STC '11)*. ACM.

Tarjei Mandt, Mathew Solnik, and David Wang. 2016. Demystifying The Secure Enclave Processor. (2016).

J.A.P. Marpaung, M. Sain, and Hoon-Jae Lee. 2012. Survey on malware evasion techniques: State of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th Inter. Conf. on*.

Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. 2010. Live and Trustworthy Forensic Analysis of Commodity Production Syst.. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection (RAID'10)*. Springer-Verlag.

Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proc. 18th Intl Symp. on Software Testing and Analysis (ISSTA '09)*. ACM.

Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. 2012. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *Proc. 2012 ACM Conf. on Comp. and Comm. Sec. (CCS '12)*. ACM.

Hyungon Moon, Jinyong Lee, Dongil Hwang, Seonhwa Jung, Jiwon Seo, and Yunheung Paek. 2016. Architectural Supports to Protect OS Kernels from Code-Injection Attacks. In *Proc. Hardware and Architectural Support for Sec. and Priv. 2016 (HASP 2016)*. ACM.

Asit More and Shashikala Tapaswi. 2014. Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing* 3, 1 (2014).

Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Annual Comp. Sec. Applications Conf.* ACM.

Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. 2016. *Taming Trans.: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory*. Springer Inter. Publishing.

Igor Muttik, Alex Nayshtut, and Roman Dementlev. 2014. Creating a spider goat: using transactional memory support for security. (2014).

Michael Myers and Stephen Youndt. 2007. An Introduction to Hardware-Assisted Virtual Machine (HVM) Rootkits. https://tinyurl.com/y8wfsye5. (2007).

Matthias Neugschwandtner, Christian Platzer, PaoloMilani Comparetti, and Ulrich Bayer. 2010. dAnubis – Dynamic Device Driver Analysis Based on Virtual Machine Introspection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Christian Kreibich and Marko Jahnke (Eds.). Lecture Notes in Comp. Science, Vol. 6201. Springer Berlin Heidelberg.

Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. 2009. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proc. 2009 Annual Comp. Sec. Applications Conf. (ACSAC '09)*. IEEE Comp. Society.

Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Sec. Symposium (USENIX Sec. 17)*. USENIX Association.

Nist.gov. 2017. National Vulnerability Database. https://tinyurl.com/yc9lbse8. (2017).

Jan Magnus Granberg Opsahl. 2013. *Open-source virtualization : Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox*. Ph.D. Dissertation.

Roberto Paleari. 2015. Fast coverage analysis for binary applications. https://tinyurl.com/y7obk3y5. (2015).

Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A Fistful of Red-pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proc. 3rd USENIX Conf. on Offensive Technologies (WOOT'09)*. USENIX Association.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. 22Nd USENIX Conf. on Sec. (SEC'13)*. USENIX Association.

Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. 2008. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. 2008 IEEE Symp. on Sec. and Priv. (SP '08)*. IEEE Comp. Society.

Michael Pearce, Sherali Zeadally, and Ray Hunt. 2013. Virtualization: Issues, Sec. Threats, and Solutions. *ACM Comput. Surv.* 45, 2 (March 2013).

Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *Proc. 4th Eur. Wksp on System Sec. (EUROSEC '11)*. ACM.

Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A Survey of Sec. Issues in Hardware Virtualization. *ACM Comput. Surv.* 45, 3 (July 2013).

Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. 2004. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proc. 13th Conf. on USENIX Sec. Symp. - Volume 13 (SSYM'04)*. USENIX Association.

Cody Pierce, Matthew Spisak, and Kenneth Fitch. 2016. Capturing 0day Exploits with PERFectly Placed Hardware Traps. https://tinyurl.com/ycrsez3y. (2016).

Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. *Measuring and Defeating Anti-Instrumentation-Equipped Malware*. Springer.

Xen Project. 2017. Xen ARM with virtualization extensions. https://tinyurl.com/k3o6h63. (2017).

Daniel Quist, Lorie Liebrock, and Joshua Neil. 2011. Improving Antivirus Accuracy with Hypervisor Assisted Analysis. *J. Comput. Virol.* 7, 2 (2011).

Nguyen Anh Quynh and Kuniyasu Suzaki. 2010. Virt-ICE: Next-generation Debugger for Malware Analysis. https://tinyurl.com/ybszcbxn. (2010).

Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. 2012. When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition. In *Proc. 28th Annual Comp. Sec. Applications Conf. (ACSAC '12)*. ACM.

Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. 2009. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. *2012 7th Intl. Conf. on Availability, Reliability and Sec.* 0 (2009).

Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proc. 11th Intl. Symp. on Recent Advances in Intrusion Detection (RAID '08)*. Springer-Verlag.

Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Syst., Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1 (March 2012).

Christian Rossow, Christian J. Dietrich, Christian Kreibich, Chris Grier, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proc. 33rd IEEE Symp. on Sec. and Priv. (S&P)*. IEEE.

Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. 2013. FireDroid: Hardening Sec. in Almost-stock Android. In *Proc. of the 29th Ann. Comp. Sec. App; Conf. (ACSAC '13)*. ACM.

Rutkowska. 2006. Subverting Vista Kernel For Fun And For Profit. https://tinyurl.com/y86ltylh. (2006).

Rutkowska. 2010. Qubes OS Project. https://www.qubes-os.org/. (2010).

Joanna Rutkowska. 2015. Intel x86 considered harmful. https://tinyurl.com/hnbulmv. (2015).

Joanna Rutkowska and Rafał Wojtczuk. 2008. Preventing and Detecting Xen Hypervisor Subversions. https://tinyurl.com/44denv2. (2008).

Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. 2014. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proc. 21st Annual Network and Distributed System Sec. Symp. (NDSS'14)*. Internet Society.

Samsung. 2017. Samsung KNOX. https://www.samsungknox.com/en. (2017).

J. Schiffman and D. Kaplan. 2014. The SMM Rootkit Revisited: Fun with USB. In *Availability, Reliability and Sec. (ARES), 2014 9th Intl. Conf. on*. IEEE.

Christian Schneider, Jonas Pfoh, and Claudia Eckert. 2011. A Universal Semantic Bridge for Virtual Machine Introspection. In *Proc. 7th Intl. Conf. on Information Syst. Sec. (ICISS'11)*. Springer-Verlag.

Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. https://arxiv.org/abs/1702.08719. (2017).

SeaBIOS. 2015. SeaBIOS. http://www.seabios.org/SeaBIOS. (2015).

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. 2007. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proc. 21st ACM SIGOPS Symp. on Operating Syst. Principles (SOSP '07)*. ACM.

Joseph Sharkey. 2016. Breaking Hardware-Enforced Sec. with Hypervisors. https://tinyurl.com/y8fuc3jg. (2016).

Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *23rd USENIX Sec. Symp. (USENIX Sec. 14)*. USENIX Association.

Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. 2009. BitVisor: A Thin Hypervisor for Enforcing I/O Device Sec.. In *Proc. ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE '09)*.

Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Comp. Sec. via Binary Analysis. In *Proc. 4th Intl. Conf. on Information Syst. Sec. (ICISS '08)*. Springer-Verlag.

Sherri Sparks and Jamie Butler. 2005. Shadow Walker - Raising The Bar For Windows Rootkit Detection. https://tinyurl.com/yag77m8y. (2005).

Patrick Stewin, Jean-Pierre Seifert, and Collin Mulliner. 2011. Poster: Towards Detecting DMA Malware. In *Proc. 18th ACM Conf. on Comp. and Comm. Sec. (CCS '11)*.

Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. 2014. *Unsupervised Anomaly-Based Malware Detection Using Hardware Features*. Springer Inter. Publishing.

Alexander Tereshkin and Rafal Wojtczuk. 2009. Introducing Ring -3 Rootkits. https://tinyurl.com/l7qnjpv. (2009).

Kevin Townsend. 2016. Mobile Malware Shows Rapid Growth in Volume and Sophistication. https://tinyurl.com/ya7ctfcz. (2016).

Petar Tsankov, Mohammad Torabi Dashti, and David Basin. 2013. Semi-valid Input Coverage for Fuzz Testing. In *Proc. of the 2013 Inter. Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM.

Jeroen van Prooijen. 2016. The Design of Malware on Modern Hardware. https://tinyurl.com/y8rwfj5t. (2016).

Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. PixelVault: Using GPUs for Securing Cryptographic Operations. In *Proc. 2014 ACM SIGSAC Conf. on Comp. and Comm. Sec. (CCS '14)*.

Giorgos Vasiliadis and Sotiris Ioannidis. 2010. GrAVity: A Massively Parallel Antivirus Engine. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection (RAID'10)*. Springer-Verlag.

Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2011. MIDeA: A Multi-parallel Intrusion Detection Architecture. In *Proc. 18th ACM Conf. on Comp. and Comm. Sec. (CCS '11)*. ACM.

Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. 2015. GPU-assisted Malware. *Int. J. Inf. Secur.* 14, 3, Article - (June 2015).

Amit Vasudevan and Ramesh Yerraballi. 2005. Stealth Breakpoints. In *Proc. 21st Annual Comp. Sec. Applications Conf. (ACSAC '05)*. IEEE Comp. Society.

Amit Vasudevan and Ramesh Yerraballi. 2006a. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. In *Proc. 2006 IEEE Symp. on Sec. and Priv. (SP '06)*. IEEE Comp. Society.

Amit Vasudevan and Ramesh Yerraballi. 2006b. SPiKE: Engineering Malware Analysis Tools Using Unobtrusive Binary-instrumentation. In *Proc. 29th Australasian Comp. Science Conf. - Volume 48 (ACSC '06)*. Australian Comp. Society, Inc.

Vassilios Ververis. 2010. *Sec. Evaluation of Intel's Active Management Technology*. Ph.D. Dissertation. KTH Information and Communication Technology.

Jack Wallen. 2016. Is the Intel Management Engine a backdoor? https://tinyurl.com/j8s2uaa. (2016).

Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2015a. Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring. In *9th USENIX Wksp on Offensive Technologies (WOOT 15)*. USENIX Association.

Jiang Wang, Angelos Stavrou, and Anup Ghosh. 2010. HyperCheck: A Hardware-assisted Integrity Monitor. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection (RAID'10)*. Springer-Verlag.

Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. 2011. Firmware-assisted Memory Acquisition and Analysis Tools for Digital Forensics. In *Proc. 2011 6th IEEE Intl. Wksp on Systematic Approaches to Digital Forensic Engineering (SADFE '11)*. IEEE Comp. Society.

Xueyan Wang and Xiaofei Guo. 2016. NumChecker: A System Approach for Kernel Rootkit Detection and Identification. https://tinyurl.com/yc5svs9m. (2016).

Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. 2015b. ConFirm: Detecting Firmware Modifications in Embedded Syst. Using Hardware Performance Counters. In *Proc. IEEE/ACM Intl. Conf. on Comp.-Aided Design (ICCAD '15)*. IEEE Press.

Filip Wecherowski. 2009. A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. https://tinyurl.com/knoms4t. (2009).

Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012b. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proc. of the 28th Annual Comp. Sec. Applications Conf. (ACSAC '12)*. ACM.

Carsten Willems, Ralf Hund, and Thorsten Holz. 2012a. *CXPInspector: Hypervisor-Based, Hardware-Assisted System Monitoring*. Technical Report. Horst Gortz Institute for IT Sec.

Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proc. 2012 42nd Annual IEEE/IFIP Intl. Conf. on Depend. Syst. and Net. (DSN) (DSN '12)*. IEEE Comp. Society.

Xiaowen Xin. 2017. Lock it up! New hardware protections for your lock screen with the Google Pixel 2. https://tinyurl.com/yb5pejys. (2017).

Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts. In *26th USENIX Sec. Symposium*. USENIX.

S. D. Yalew, G. Q. Maguire, S. Haridi, and M. Correia. 2017. T2Droid: A TrustZone-Based Dynamic Analyser for Android Applications. In *2017 IEEE Trustcom/BigDataSE/ICESS*.

Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining Hardware Virtualization and Softwareemulation for Transparent and Extensible Malware Analysis. In *Proc. 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments (VEE '12)*.

Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. 2011. Sec. Breaches As PMU Deviation: Detecting and Identifying Sec. Attacks Using Performance Counters. In *Proc. 2nd Asia-Pacific Work. on Syst. (APSys '11)*. ACM.

Fengwei Zhang. 2013. IOCheck: A framework to enhance the security of I/O devices at runtime. In *2013 43rd Annual IEEE/IFIP Conf. on Depend. Syst. and Net. Wksp (DSN-W)*.

F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *2015 IEEE Symp. on Sec. and Priv.* IEEE.

Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. SPECTRE: A Depend. Introspection Framework via System Management Mode. In *Proc. 43rd Annual IEEE/IFIP Intl. Conf. on Depend. Syst. and Net. (DSN) (DSN '13)*. IEEE Comp. Society.

Fengwei Zhang and Hongwei Zhang. 2016. SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Sec.. In *Proc. Hardware and Architectural Support for Sec. and Priv. (HASP)*. ACM.

Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo. 2013. DEMO: Enabling Trusted Stores for Android. In *Proc. of the 2013 ACM SIGSAC Conf. on Comp. &#38; Comm. Sec. (CCS '13)*. ACM.

Received Date 1; revised Date 2; accepted Date 3

Appendix for the paper: Who watches the watchmen:

Marcus Botacin[1], Paulo de Geus[2], André Grégio[3],
(1) University of Campinas
Email: marcus@lasca.ic.unicamp.br
(2) University of Campinas
Email: paulo@lasca.ic.unicamp.br
(3) Federal University of Paraná
Email: gregio@inf.ufpr.br

In this section, we show an overview of the presented technologies, tools, and solutions, aiming to ease comparison among them. As the tabulated items are themselves related in two distinct ways—by the employed technology and by the solution goal—we present both, so a given solution may appear more than once.

Table I presents the HVM-based tools and solutions.
Table II presents the SMM-based tools and solutions.
Table III presents the privileged rings-based tools and solutions.
Table IV presents the hardware-based tools and solutions.
Table V presents the performance counters-based tools and solutions.

Table I: Summary of HVM-based tools and solutions.

Tool	Purpose	Target OS	Technology	Hypervisor	Resources
BluePill	offensive	Windows Vista	AMD SVM	–	network backdoor
HVM Rootkit	offensive	Windows XP	AMD SVM	–	–
Ether	malware analysis	Windows XP	–	Xen	syscall tracer, unpacker
CXPInspector	malware analysis	Windows 7 x64	Intel VT-x	KVM	memory tracking, profiling
MAVMM	malware analysis	Ubuntu Linux	AMD SVM	own hypervisor	syscall tracer, unpacker
HyperDBG	debugging	Windows XP	Intel VT-x	–	kernel debugger, graphical user interface
SPIDER	debugging	Windows XP Ubuntu Linux	–	KVM	trap flag hider, unlimited breakpoints
V2E	execution replay	Linux Windows XP	HVM DBT	TEMU	transparent collection, execution replay, emulated instruction changes, emulated page table translator
Kang et al. 2009	execution replay	Windows XP	HVM DBT	Ether TEMU	transparent collection, execution replay, dynamic state modifications
BitVisor	policy enforcement	Windows XP Windows Vista Linux	Intel VT-x	–	para-passthrough, I/O policy, DMA MITM
SecVisor	attack prevention	–	AMD DEV	–	code execution policy, code authorization, kernel-userland flow integrity
HyperSleuth	forensics	Windows XP	Intel VT-x	–	syscall tracer, dump on write, lie detector
Quist et al 2001	unpacking	Windows XP	Intel Vt-x	Ether	binary section and header rebuilding, VAD import rebuilding, AV submission

Table II: Summary of SMM-based tools and solutions

Tool	**Purpose**	**Target system**	**Trigger**	**Resources**
Duflot et al. 2007	offensive	OpenBSD	—	unrestricted physical memory access
Embleton et al. 2008	offensive	–	keyboard IRQ on APIC chipset	keylogger
Schiffman and Kaplan 2014	offensive	Linux	USB-PS2 emulation handling rerouted from USBHC	keylogger, UDP transmission
MALT	debugging	Windows, Linux	performance counter overflow	register access, memory inspection, step-by-step, BIOS flashing, GDB integration
SMMDumper	forensics	–	APIC-redirected known key-pressed sequence interrupt	memory dump, UDP packets, code callbacks
Wang et al. 2011	forensics	–	IPMI	memory dump, consecutive snapshots, PCI DMA, SMM-based semantic gap bridging, BIOS NIC driver
SPECTRE	attack detection	Windows, Linux	periodic timer	memory pattern matching, BIOS NIC heartbeat
I/O Check	I/O integrity	–	–	APIC remapping check, NIC firmware hash check
HyperSentry	hypervisor integrity	Intel VT-x, Xen Hypervisor	IPMI, performance counter overflow	hash-based integrity check, SMM-based semantic gap bridging, root-non-root transition bridging
HyperCheck	hypervisor integrity	QEMU, real NIC, Xen hypervisor, Linux, Windows	–	SMM-based semantic gap bridging, hash-based integrity check

Table III: Summary of privileged rings-based tools and solutions.

Solution Class	**Ring**	**Underlying Technology**	**Capabilities**	**Limitations**
HVM	-1	extended processor instruction set	attack: hypervisor attacks, defense: kernel/userland monitoring	hypervisor rewriting, semantic gap, considerable overhead
SMM	-2	system BIOS	attack: boot attacks, defense: kernel, userland and hypervisor monitoring	BIOS rewriting, semantic gap, locked BIOS
AMT	-3	Chipset	attack: whole system view, defense: kernel, userland, hypervisor and BIOS monitoring	chipset dependent, semantic gap
SGX	–	Processor Enclave	attack: malware integrity check, defense: running software cannot be monitored	API-dependent code
hardware	–	physical external hardware	attack: –, defense: tamper-proof monitoring	hardware development efforts, bus monitoring rate, semantic gap
performance counter	–	processor feature	attack: side-channel detection, defense: low-overhead tracing and profiling	no process isolation, kernel mechanism configuration
GPU	–	PCI Card	attack: DMA snooping, defense: DMA monitoring	semantic gap, DMA blocking
TSX	–	concurrency control hardware	attack: –, defense: commit-based control flow	limited block size

Table IV: Summary of hardware-based tools and solutions.

Tool	**Technology**	**Data Collection**	**Filtering**	**Vulnerabilities**
Copilot	PCI Card	snapshot	No	timing, dynamic state modification
SnapMon	SOC	snapshot	no	timing, dynamic state modification
SnoopMon	SOC	snooping	yes	–
Ki-Mon	SOC	snooping	yes (Hardware-Assisted Whitelisting)	–
Kargos	–	snapshot	–	–

Table V: Summary of performance counters-based tools and solutions.

Tool	Class	Type	Vendor	Purpose	Intel.	Intro.	Tools	Underlying Solution	Injection	Limits
Branch Trace	branch monitor	—	—	delusion attacks, ROP	root cause analysis, CALL-RET CFI	Windows debug symbols	shellcode extractor, ROP detector	CWXDetector	—	no CG or CFG generated
CFIMon	branch monitor									
KBouncer	branch monitor	LBR	Intel	ROP detector	CALL-RET CFI	–	process blocking	–	yes	LBR gadget-length, code injection
ROPecker	branch monitor	LBR	Intel	ROP detector	CALL-RET CFI	–	—	—	—	LBR gadget-length, static code database
Pierce et al. 2016	events monitor	—	—	ROP detector	baseline					
Kompali and Sarat 2014	events monitor	–	Intel	abnormal behavior	baseline	N/A	CPU usage, mem usage, cache usage, branch prediction	Vtune	–	no process information, Vtune limits
HPCHunter	events monitor	–	–	abnormal behavior	baseline + SVD/SVM					
Yuan et al. 2011										
Demme et al. 2013										
Wang and Guo 2016										
Malone et al. 2011										
Tang et al. 2014										
Confirm										

Table VI presents protection mechanisms employed by the presented solutions.

Table VI: Protection mechanisms used by overviewed solutions.

Solution Class	**Mechanism**	**Protection**
HVM	trap flag	exception handler
HVM	loader driver	rootkit hider technique
HVM	hypervisor code	page fault handler trap
HVM	timing	TSC change
SMM	BIOS change	online BIOS flashing
SMM	timing	TSC change
performance counter	MSR disabling	periodic kernel checking
DMA	DMA blocking	device exclusion vector

Table VII presents a comparison of solutions according to their purposes.
Table VIII presents a comparison of solutions according to their overhead.
Table IX presents a comparison of solutions according to their required development effort.

Table VII: Comparison of solutions according to their purposes.

Purpose	Technology	Advantages	Disadvantages
offensive	HVM	late launch	–
offensive	SMM	–	BIOS locking
offensive	SGX	remote attestation	–
malware analysis	HVM	late launch	hypervisor rewriting, overhead, semantic gap
malware analysis	SMM	–	BIOS rewriting/locking, semantic gap
malware analysis	performance counter	low overhead	limited data capture, limited process information
malware analysis	GPU	low overhead	limited to DMA data, DMA blocking, semantic gap
debug	HVM	easy register access	
debug	SMM		limited to SMI
attack detection	HVM	–	vulnerable to hypervisor attacks
attack detection	SMM	inspect hypervisors	have to implement network support
attack detection	GPU	low overhead	DMA-limited
attack detection	PCI-card	low overhead	DMA-limited
attack detection	hardware	tamper-proof	no active component
policy enforcement	HVM	IOMMU	–
integrity check	HVM	kernel monitoring	hypervisor attacks
integrity check	SMM	hypervisor check	coupled semantic gap
integrity check	DMA	low overhead	timing attacks
side effects	event counter	low overhead	limited process isolation
ROP	branch monitor	runtime code monitor	increased overhead
ROP	event monitor	side effects detection	limited process information

Table VIII: Comparison of solutions according to their overhead.

Technique	Overhead	Reason/Limitation
HVM	high	single-step trap at hypervisor level
SMM	high	single-step trap at BIOS level
performance counter	medium-low	branch-step trap at kernel level
GPU DMA	near-zero	GPU blocked for other calculations
dedicated PCI DMA	near-zero	specific purpose
external hardware	zero	no interruption/interference

Table IX: Comparison of solutions according to development effort.

Technique	**Development effort**	**Reason/Limitation**
HVM	high	hypervisor rewrite
SMM	high	BIOS rewrite
external hardware	high	hardware project
dedicated PCI DMA	medium	device driver
performance counter	medium	ordinary kernel driver
GPU DMA	low	GPU program