HARES: Hardened Anti-Reverse Engineering System

Technical Whitepaper

Jacob I. Torrey
Assured Information Security, Inc.
torreyj@ainfosec.com / @JacobTorrey

ABSTRACT

This paper provides a technical overview of the HARES software protection research effort performed by Assured Information Security. HARES is an anti reverse-engineering technique that uses on-CPU encryption [7] in conjunction with Intel x86 TLB-splitting [12] in order to significantly increase the effort required to obtain the clear-text assembly instructions that comprise the target x86 application.

Performance and use-cases of the system are presented, and a number of weaknesses and future works are discussed. Related works are compared and contrasted with HARES in order to highlight its improvements over the current state-of-the-art.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*System architectures*; D.2.0 [**Software Engineering**]: General—*protection mechanisms*

Keywords

Security, Obfuscation, Encryption, Split-TLB, AES, TRESOR

1. INTRODUCTION

This paper provides details on the design and implementation of the research effort HARES. HARES provides a proof-of-concept capability to transparently execute fully-encrypted binaries on a standard Intel CPU with minimal performance impact.

There are a number of use-cases for such a capability: protection of algorithmic IP, providing resistance to mining for ROP gadgets and significantly increasing the difficulty of "weaponizing" a crash-case into a RCE. Background is provided, followed by details on the HARES design, and an evaluation of the proof-of-concept. Finally related works and conclusions are presented.

2. PROBLEM STATEMENT

The current state-of-the-art for program protections are vulnerable to sufficiently motivated attackers and reverse-engineers. Resulting from these weaknesses, sensitive algorithms can be copied, and vulnerabilities in software exploited.

2.1 Background

This research builds upon numerous past research efforts and technologies, the following subsections provide the requisite background for a generally technical reader to fully understand the methodology of HARES.

2.1.1 Intel x86 Architecture

HARES supports the modern Intel x86 CPU architecture (Core-i 4000 series and newer), a modified-Harvard CISC instruction set architecture commonly found on today's desktop and laptop computer systems. The following paragraphs present a deep-dive into certain features of the x86 architecture which are leveraged by HARES in order to provide anti reverse-engineering protections.

Privilege Rings. Initially implemented in the Intel 80286, and extended more fully with the 80386, privilege rings and "protected mode" provide a level of protection from misbehaving applications. Prior to protected mode, the CPU operated in "real mode", where every application had full access to the entire memory space, and the ability to directly communicate with hardware. If a real mode application were to overwrite a critical portion of memory storing the operating system (OS), there were no protective mechanisms in place to prevent it from doing so.

In protected mode, the OS kernel could load itself into a more privileged "ring" than the applications it was supporting and limit the applications' ability to impact the system as a whole. There are four "official" privilege rings $(0-3)$ and two or three other modes of execution that are typically referred to as rings $-1, -2, \& -3$. Modern OSs typically utilize only two of these rings: ring-0 for the privileged "kernel-mode" OS routines, scheduler and interrupt handlers and ring-3 for the "user-mode" applications where they are limited in their abilities to execute certain instructions, access protected memory regions, interact with the hardware, generally to impact the system as a whole [5].

As the x86 architecture has evolved and the scrutiny placed on the low-level implementations increased in the security community, two more CPU modes have been informally called rings: hypervisor (VMM) root mode (ring -1) and system management mode (ring -2). Ring -3 is also used when referring to Intel's Active Management Technology (AMT), but is not relevant for this research. Ring -1 is more privileged than the OS kernel, and is used by hypervisors to virtualize multiple OSes without requiring modification to the OS kernel (full-virtualization). The specifics of ring -1 are discussed later as HARES is implemented as a thin-hypervisor.

Ring -2, more commonly known as system management mode (SMM), is a parallel and hardware-enforced stealth mode of execution. It was originally developed for BIOS and chip-set manufacturers to run platform-specific routines for power and thermal management without exposing the internals to the OS. When the CPU is executing in SMM, it has a full view of memory, while when executing in protected mode, the chip-set blocks access to SMM memory regions (SMRAM). In order to switch from protected (or real mode) into SMM, a system management interrupt (SMI) is issued to the CPU and the hardware saves all CPU context, allowing the SMM code to execute without the OS or hypervisor detecting it was interrupted. The conditions for the hardware to generate an SMI are defined via a number of chip-set registers that can be locked during SMM initialization [5].

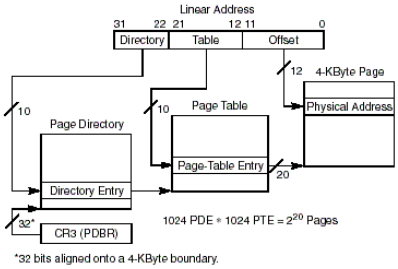

Figure 1: Paging Translation on x86[1]

Paging & Memory Management. Modern CPUs provide the ability to provide each process with a unique view of memory [5]. This feature, known as virtual memory, eases the OS's task of isolating different applications and providing each application with a different view of memory. When a virtual memory address is accessed by an application, the CPU uses a number of data structures to automatically translate the virtual address into a physical address. This process, outlined in Figure 1, uses the CPU's `CR3` register to find a page directory and optionally a page table that holds the physical address. In most modern OSes, each process is given its own set of page translation structures to map the 4GiB flat memory view provided to the system's physical memory (assuming a 32-bit OS).

Intel VT-x, VT-d & EPT/VPID. A growing trend in information technology is the use of virtual machines, or VMs to enable data-center consolidation. A hypervisor, or virtual machine monitor (VMM), allows multiple OSes to run simultaneously on the same physical system, each isolated from the others and provided with the appearance of a normal system environment. While some hypervisors require

[1]Image from `http://viralpatel.net/taj/tutorial/image/paging.gif`

changes to the guest OS (para- virtualization) to function properly, many leverage newer CPU extensions to allow an unmodified OS to run with only minor interactions from the hypervisor. These extensions, known as virtual machine extensions, or VT-x on Intel, improve performance by empowering the CPU and chip-set to perform more of the isolation and VM memory management in hardware as opposed to software. VT-x allows the hypervisor to set a number of different exit conditions for each guest VM that, when met will trigger a VM Exit, returning control to the hypervisor for processing.

After the initial release of VT-x, it was discovered in [15] that it was possible to use a device's direct memory access (DMA) capabilities to bypass VT-x's protections and compromise the hypervisor. In response to this weakness, Intel released the extensions for directed IO (VT-d), providing a memory management unit (MMU) for IO DMA memory accesses, preventing a device from directly accessing main memory outside of its policy-proscribed region(s).

In the latest version of hardware virtualization extensions, Intel and AMD have released the extended page table (EPT) or rapid virtualization indexing (RVI) technologies, respectively. These allow the hypervisor to take even less of a role in the memory management and isolation of each guest by providing another layer of paging structures to translate the physical addresses that the VM OS believes to be the physical address (guest physical address) to the machine physical address. The CPU can automatically translate these in a similar fashion to conventional paging and provide a VM Exit, analogous to a page fault, for the hypervisor. These translations are stored in the TLB, and tagged with each guest's VM process ID (VPID) so they need not be flushed on VM context switch.

These aforementioned technologies significantly aid the hypervisor in running multiple VMs in an isolated fashion with relatively minor performance impacts. The implementation in Section 3 extensively leverages these technologies to perform the TLB-splitting for user-land Windows applications.

Translation Lookaside Buffer (TLB). The translation lookaside buffer, or TLB, acts a cache for these paging translations. Due to relatively high memory latency compared to cache-access speed, a page translation look up is expensive in terms of time; therefore, these operations are optimized by caching the virtual-to-physical mappings in the TLB. While logically, the TLB stores the translations for all accessed addresses in the same region of silicon, the physical implementation splits the TLB (Figure 2) into two: one for instruction addresses (I-TLB) and one for data addresses (D-TLB). This implementation detail is important, as it allows the TLB to point to different addresses for instruction fetches as compared to data accesses.

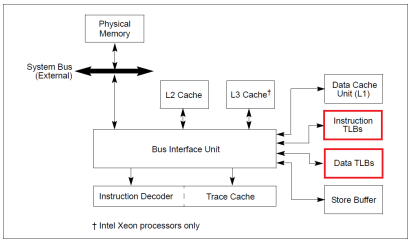

Figure 2: Core 2 and Previous CPU Architecture (TLBs Highlighted)[3]

In earlier processors[4], the TLB was a single-layer (L1) cache that, in the silicon was split into two: a data TLB (D-TLB) and instruction TLB (I-TLB), for data and instruction fetches respectively. With the release of the Nehalem architecture[5], a second layer (L2) cache was introduced, the shared TLB (S-TLB) that provides a larger cache for both the I-TLB and D-TLB in the event of an eviction from the L1, further reducing TLB "misses" and the resultant performance hit.

The TLB operates in a similar fashion to the regular CPU caches (L1, L2 & L3) in that it typically does "the right thing", requiring minimal manual management. The Intel architecture does provide a small number of instructions for flushing the TLB, either a single entry or multiple entries. The TLB is flushed (excepting for pages marked "global" in the paging structures) when the value of the `CR3` CPU register is changed (typically when switching

[3]Image from Intel Software Developer Manual 3A [5]
[4]Up to and including Core-2
[5]The first Core-i series processor micro-architecture

processes). With the addition of EPT and VPID, other instructions that operating in a similar fashion were introduced to managed a VM-aware TLB.

2.1.2 TLB-Splitting

There has been some past work that took advantage of this split-TLB nature for both defensive and malicious purposes in the past. In the PaX/GRSecurity project, whose aim was to harden the Linux OS from attack, the no-execute (`NX`) bit was emulated by overloading the user-supervisor bit (`U/S`) and splitting the TLB to prevent instruction fetches to a protected page [9]. On the offensive side, the Shadow Walker root-kit [11] built upon this work as well as work done to prevent self-verifying applications from detecting corruption [14]. Shadow Walker is designed to hide the presence of a malicious kernel driver through TLB-splitting. When this driver is accessed as data, such as by an antivirus tool, Shadow Walker points the D-TLB towards the unmodified kernel region, thus hiding the compromise. When the target section is executed, the I-TLB is filled with the address of the malicious driver's code, allowing it to run as expected.

In [12], the new Nehalem (Core-i series) microarchitecture is discussed and highlights the addition of a new shared TLB (S-TLB). With this new second level cache for address translations, previous works no longer were able to maintain the split-TLB. The MoRE work provided the capability, through the use of the more granular permissions afforded by EPT to once again simulate this split TLB environment, with minimal performance impact.

AES-NI Instruction Set. In the aftermath of [13], Intel developed a hardware-based AES implementation in their more recent CPU offerings (Core-i series 4000 and new) in order to increase the challenge of side-channel attacks. The instructions to utilize this implementation were collectively referred to as AES-NI, or the Advanced Encryption Standard-New Instructions [5]; AES-NI provides an improvement in security as well as a performance enhancement from being hardware-accelerated.

2.1.3 TRESOR and On-CPU AES

In [7], the new AES-NI instructions were used in a Linux kernel patch to provide on-CPU encryption and decryption support where the key was protected from cold-RAM attacks[6]. TRESOR used the CPU debug registers (`DR0-7`) as a key storage mechanism and used its position in the kernel to prevent applications from accessing those registers[7]. TRESOR would have the user input the key into the CPU debug registers during early boot to limit the number of running applications that could intercept the key.

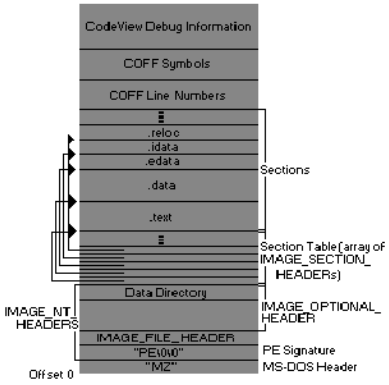

Figure 3: Windows PE File Structure[8]

2.1.4 Windows PE Format

Modern Windows OS applications and kernel drivers are binaries in a format known as portable executables (PE) (Figure 3). This format provides the application or driver loader with the requisite information to load it into memory, adjust any addresses which may have been changed and link in any shared libraries before execution. It divides the binary into a number of different sections, some for code (generally labeled '`.text`'), some for data ('`.data`') and other informational sections. Of these extra sections, the most important to this effort is the '`.reloc`' section, which lists the number and location of addresses which must be updated at load time if the compile-time address is different from load address.

[6]When RAM is brought to a very low temperature, it will persist its contents for a period of time, allowing an attacker to insert the RAM into another machine to dump the contents of memory

[7]Debugging applications like GDB will automatically switch to using software breakpoints if the kernel reports no free debug registers

[8]Image from `http://i.msdn.microsoft.com/dynimg/IC155437.gif`

2.2 Research Question

The goal of the HARES research was to determine if a TRESOR-like system implemented in conjunction with a TLB-splitting capability would allow for the apparent execution of fully AES-encrypted applications without requiring source for that binary. If possible, what would the limitations, security weaknesses and performance impacts on such a system.

3. PROPOSED SOLUTION

The below subsections describe HARES, described in components that together comprise the research prototype. The ordering of the following subsections are in line with how the research effort progressed, and should provide the reader with a picture of the "journey".

3.1 AES-NI in VMM

In order to minimize the duplication of effort, HARES was based upon the open-source VMM TLB-splitting capability provided by MoRE [12]. MoRE was a thin-VMM for Windows 7 (32-bit, no PAE) that added the requisite functionality to simulate a split-TLB to the 'vmcpu' root-kit. In order for HARES to protect the AES-NI keys from a compromised OS kernel and to integrate smoothly with the MoRE code, the on-CPU AES-NI routines (using the debug registers) from TRESOR were "hoisted" to the VMM and the protections against an OS or application from reading the registers were moved to the VMM layer. This was done by adding debug register accesses (read and write) to the VM Exit conditions in the VM control structure (VMCS) for the VM. The VMM handler for these exits would either return a zero value for reads to all registers or silently discard writes[9].

As MoRE was initially developed for operation in 32-bit mode, only 128-bit AES keys are supported. The authors however see no barrier to adding support for 256-bit key lengths on a 64-bit version of HARES.

3.2 AES Encryption of PE Executables

Once the HARES VMM could perform on-CPU routines for AES decryption, a tool was created to take as input a PE .exe file, and output it with all the sections marked as executable AES encrypted. This process required parsing the PE structures in order to locate all the code sections in the binary, leaving the meta-data structures and data sections untouched. In Section 4.2.1, we discuss the challenges that arose during this process, and how this is a extremely difficult operation to perform correctly. In brief, many compilers will compress the output binary by interleaving code and data into the same section, thus when run by HARES, the application would not function properly as certain data structures were encrypted and accessed as data. We will discuss our current work-around for this issue in Section 4.2.1.

[9]In future work, we anticipate engineering a more compatible solution that does not interfere with application debugging operation

3.3 Windows Process Creation Monitoring

The HARES VMM is launched through a co-operative Windows kernel driver, which via hyper-calls, can interact with the VMM from ring-0. This driver registers a new callback for process creation: configuring the OS to notify the driver every time a new application is started. If the newly-started application is not encrypted, the driver will perform no action, allowing the application to operate as usual.

Once a HARES-protected application is started, the driver will perform a number of actions in order to allow the OS to seamlessly execute the program. A limitation of the current prototype is the requirement to lock the program's encrypted pages into non-paged memory. This is needed as OSes will page out unused memory pages to disk until they are needed in order to free up RAM for other processes. Since the goal of HARES is to act transparently to the OS and user, if the OS were to page out an application's page and replace it with memory from a different process, the EPT mapping would still reflect the HARES application and corrupt the process (the OS's write to that page would update the data page, but instruction fetches would still go to the original program's page). In order to combat this, the current HARES prototype will lock the pages into non-paged memory. This is a constraint on the system's resources as the non-paged pool is typically in high-demand and much smaller, so the application size for this prototype is limited.

There are engineering solutions to bypass this limitation, if Microsoft were to register a new callback for a region of memory being paged in or out,

HARES could update the protected pages accordingly. Alternatively, it would be feasible for the VMM to monitor modifications to the target process' paging structures and infer the same information in a more "brute-force" method.

Once the memory has been locked, it can then be decrypted by VMM (on-CPU using AES-NI) into a region of memory protected by the hypervisor as execute-only.

3.3.1 Simulating the Split-TLB

On pre-Nehalem systems (Core 2 and previous), the I-TLB and D-TLBs were not directly connected, and thus a full split was possible. With newer systems, the interaction with the S-TLB requires the use of EPT's more granular permissions to prevent the "pollution" of the incorrect entry being merged across from the I-TLB to the D-TLB or vice-versa[4].

HARES uses a similar model as Shadow Walker for emulating a Harvard architecture on x86 [11]. This work configured the VMM to trap on EPT faults (similar to a page-fault in the kernel) and will then use the more granular permissions available (E/R/W) in the EPT paging structures to direct differing types of memory fetches to the correct location in physical RAM.

EPT Fault Handling. When the VMM is launched, it creates an identity map, mapping each guest-physical page to the same machine-physical address with every mapping present and full permissions. Through the use of the larger page sizes available to EPT, this is a relatively small page table configuration. Once the application is loaded into memory by the OS, the VMM is given the list of memory addresses corresponding to the code sections to split and the EPT paging structures are updated to map 1-to-1 for each page. These entries are then either mapped execute-only to the decrypted code pages or read/write to the encrypted memory.

When a memory fetch occurs and there is no matching entry in the TLB, the CPU will examine the paging structures, if the EPT mapping currently set is not appropriate for the type of fetch, a fault will occur. The HARES VMM will determine the type of fault (instruction or data) and update the EPT paging structure for that memory page to point to the proper region of physical memory and the permissions set to allow the fetch. The guest is then resumed and execution continues until another fault is triggered.

COW Detection and Support. Windows provides the feature of copy-on-write to minimize memory duplication when multiple instances of the same application are run in parallel; the same static code pages are all referenced from all of the duplicate programs. If a single instance of a program makes a modification to the code page(s), the OS will perform this copy-on-write operation. It was identified in [12] that this process would move the active application pages elsewhere in memory, defeating the TLB-split. In order to keep abreast of these changes, the VMM is configured to trap on the OS process switch (change of the `CR3` register) and will re-scan the application's paging structures to identify any modifications and update the list the VMM uses for managing the TLB-splitting operation.

3.4 Secure Tear-Down

The OS driver is also notified by the kernel of programs terminating via the same mechanism as process creation monitoring. When the HARES driver's callback is executed, it issues a hyper-call to the VMM to overwrite the decrypted code page with NULL-bytes and cease the TLB-split for that application.

4. RESULTS EVALUATION & DISCUSSION

The below subsections provide a brief overview of the results of evaluating the prototype, and a discussion of its benefits and possible attack vectors against a HARES-protected application.

4.1 Protections Afforded

The protections provided by HARES are: the protection of a program's code (data is still vulnerable) against static analysis and significantly raising the bar for dynamic reverse-engineers. These protections can help prevent sensitive algorithms and IP from being reversed as well as protect a vulnerable application against certain types of exploitation. Due to the Harvard nature of the HARES run-time environment, any code-injection attacks would be diverted to the encrypted data pages, rendering

them useless. Mining the encrypted application for ROP gadgets, or "weaponizing" a crash-case into a RCE would also be significantly more challenging.

4.2 Performance & Compatibility

The following paragraphs, the performance of the prototype will be detailed. These results provide a rough estimate of how a HARES system would function, performance will certainly change for differing applications and if any engineering enhancements performed.

Tested Applications. The HARES team utilized the synthetic test-suite used in [12] in order to determine the performance impact of running a HARES-protect application. In order to determine the feasibility of more generalized use, HARES was used to encrypt the following three Windows 7 utilities (highlighting that HARES does not require source): Notepad, Calculator and Paint. These small applications could run with the memory constraints imposed by the proof-of-concept and were used to gather ground-truth into how HARES would impact the usability of a typical GUI program.

Performance Impact. Naturally, since HARES creates a duplicate page for each executable page of the target program, there is increased memory usage, at the worst-case (every section is marked as executable), the memory footprint is slightly less than doubled. The performance impact is primarily noticeable increased start-up time. Overall (including start-up time), the performance impact was comparable to the results from [12], or about 2%. In terms of responsiveness for a GUI application, the presence of HARES is unnoticeable to the end-user.

4.2.1 Application Support Challenges

One of the most difficult challenges with the research effort was to encrypt binaries without access to the source code or the ability to recompile the program with control over the linking and output layout. In many cases, the compiler will "compress" a code and data section by combining them into a single, dual-use section. During execution, when the program is run under HARES, the data fetches accessing strings, or other data values in the code section were encrypted, even to the program's code itself. This resulted in program instability or the encrypted data being used incorrectly.

A work-around for cases where recompilation is not possible is to put HARES in "learning" mode and run the program unencrypted. After exercising the program (manually in this effort, in the future with symbolic/concolic execution), HARES would output memory regions used as data in an executable section. This output could be passed to the PE encryption tool to pass over these regions in the binary, leaving the areas of the program accessed as data in the clear to enable smooth execution.

4.3 Security Weaknesses

As there are no silver-bullets in security, the following subsections outline certain classes of attack as well as considerations for key management and deployment strategies.

4.3.1 Attack Scenarios

The below paragraphs describe the different scenarios in which it is imagined a HARES-protected application would be subjected to attack. As is always the case in security, there are sure to be additional scenarios not conceived nor described in this paper.

Theft of Protected Application. The simplest attack scenario would be an attacker copying an application from an authorized computer to another system to commit software piracy or examine the application for vulnerabilities. In this scenario, the PE file is encrypted, as well as a memory dump of the PE image, thus the attacker will be unable to decrypt the application on a non-authorized system save by brute-force attacks on the AES key.

Compromise of Authorized Host. This scenario is when an attacker successfully gains remote access to an authorized system and can launch software-based attacks against the HARES system. This attacker is assumed to have OS (ring-0) privileges, though is unable to penetrate the VT-x barrier save by an attack against the boot process, to take effect on the next reboot.

"Authorized" Local Usage. The final scenario in scope of this white-paper is when an attacker has complete, physical access to an authorized device,

and is able to perform physical attacks against the hardware. This is the most difficult attack to defend against; in the following subsection, physical attacks are discussed specifically.

4.3.2 Physical Attacks

If an attacker were to gain physical access to a computer system currently executing (i.e., authorized) a HARES-protected application, there are a number of possible attack vectors available to the attacker in order to reduce the protections afforded by HARES. This class of attack is subdivided into the following three sub-classes viz.: hardware debugging, memory dumping and side-channels. Each is described below in more detail.

JTAG/XDM. The barrier to entry for hardware debugging on x86 has dropped significantly in recent years, with costs dropping from five-figures to less than $1000. JTAG is simply a wire-protocol, thus not every "JTAG debugger" on the market can debug an Intel x86 platform. The use of these devices are physically invasive, generally sitting inbetween the CPU and the motherboard, requiring the system to be rebooted (removing the key from memory) under attacker control. If the HARES system has the key stored locally, and has a policy in place to allow it to be loaded into the CPU debug registers at early boot, it is possible an attacker would be able to exfiltrate the key from the CPU registers. However, if the system requires an authorized user to boot the system (e.g., via a BIOS password or FDE key), the attacker would only succeed in performing a DoS of the target system. It is for this reason that the authors suggest complementary protections put in place to follow best-practices for network endpoints.

Cold-RAM & Other RAM Dumping Techniques. The current implementation of HARES is vulnerable to a cold-RAM attack targeting the decrypted instructions, the TRESOR-based on-CPU AES will prevent the key from theft. In Section 4.4, we discuss how the application's decrypted instructions can be stored only in the CPU cache and prevent this weakness. Other RAM dumping techniques, such as bus-snooping, etc. would operate in a similar fashion, the current prototype could be circumvented to obtain the decrypted program, but the key would be protected. The on-CPU storage of the decrypted instructions would be able to protect against these other methods of RAM exfiltration.

Side-Channels. There are certain to be system-wide impacts of HARES that a determined attacker would be able to utilize to gain further information about a protected application. While the majority of these are outside the scope of this white-paper, a simple "instruction inference engine" could be created to single-step through the HARES-protected program and probabilistically determine the last-executed instruction from the impact it had on the system and other information from the system (e.g., performance counter MSRs).

4.3.3 Key Loading & Distribution

While the current HARES prototype research effort aimed to specifically focus on the protective capabilities available on existing systems, and key management was left to future work, the following paragraphs outline some possible key loading options.

Key Loading. In [7], the AES key is loaded into the CPU's debugging registers early in the OS's boot process, when the majority of applications found on a typical system have not yet been started. This concept of minimizing the number of applications running when the key is loaded is a sound way to reduce risk. In order to augment the user experience, the key could be stored on a USB device (e.g., a YubiKey[10]).

For more security-conscious environments, the key could be sealed via the TPM such that it is only unsealed if the system is in the proper state, and the HARES VMM has been loaded as the measured launch environment. On Intel vPro systems, this provides a seamless user experience and significant security improvements as the key will never be released if there is a malicious hypervisor rootkit attempting to circumvent the HARES system. Once the HARES VMM has been loaded, and the key inserted into the CPU registers, they can be wiped from memory and the TPM PCRs extended with random data in order to prevent another application from unsealing the key(s).

[10] `https://www.yubico.com/products/yubikey-hardware/yubikey-2/`

On systems without Intel TXT/vPro, systems such as [6] can be utilized to perform similar verification of a platform's state prior to loading a key from a protected storage mechanism. The use of these techniques would prevent the HARES key from being released unless the dynamic verification succeeds, implying there is no malicious VMM or attacker attempting to reverse engineer or debug the key loading process.

Protecting the Key. Once the key has been loaded into the CPU registers, it must be erased from cache and main memory, it is important to ensure the key is not resident in memory if a malicious process is able to prevent the CPU cache from being written back to RAM (such as with the `INVD` instruction). The HARES VMM configures the system to trap on accesses of the debug registers and will ensure NULL is returned for all reads and writes are silently discarded. The advantage of doing this in the VMM versus the OS driver is that a malicious OS, or kernel-mode driver will still be unable to read the key from the CPU registers.

With the VMM protecting the key in the CPU registers, it is still vulnerable to attacks from a malicious system management mode (SMM) handler. Protecting against SMM-based threats was outside of the scope of this initial effort, but may be addressed in future works.

4.3.4 Emulation & Virtualization Attacks

If the HARES VMM can be induced to run in a nested virtualization environment, or under full emulation, the CPU debug registers can be read and the protections bypassed. For this reason, it is critical to use a secure loading mechanism such as Intel TXT or a software-based equivalent, such as [6] in order to prevent HARES from being used if there is already a more privileged hypervisor or debugging environment running on the system.

Emulation & VMM Detection Techniques. Intel TXT and the software-based Conqueror [6] are able to either replace or detect the presence of a hypervisor. In [10], it is shown that certain instructions sometimes reveal the presence of a VMM, as well as the clock-skew introduced by the increased overhead required to context-switch on a VM Exit. If a trusted clock is available, then detecting the presence of emulation or a malicious VMM is possible, and key not loaded, thus protecting it from attack.

4.4 Future Enhancements

The HARES prototype was developed as a proof-of-concept, designed to test the feasibility of such an approach and provide initial estimates of the performance impacts on the protected program and system as a whole. The HARES research team is exploring some additional research and engineering capabilities using the HARES prototype as a foundation.

Engineering Improvements. The current prototype requires substantial improvements in order to render it usage in a typical enterprise environment. A number of limitations would need to be removed, namely with the memory-management of a protected application. Currently the program is locked into the non-paged memory pool, of which there is a limited supply. Additionally, due to certain hard-coded values, the proof-of-concept will not function on systems with greater than 2 GiB of memory, as the Windows kernel memory layout changes; a more flexible system will have to be implemented in order to operate as a fielded product.

SMM Protections. System management mode attacks are not new in the public art, though until recently were dismissed by many as too targeted in order to pose a real threat. In [8] and [16], these assumptions were challenged, showing the introspective capabilities of SMM as well as the scale of possible vulnerabilities.

Future work in this area is needed; there is a significant need for a defensive mechanism in order to protect the VMM, OS and user applications from a rogue SMM handler. Intel's solution of a dual-monitor mode [5], may be the ultimate goal, but it has yet to materialize with high market penetration. In the meantime, performing research into the detection of a malicious SMM would prove valuable simply as a method-of-last-resort, changing systems if a compromise is detected. There is some existing work in this field [3], but future work should build upon this to ensure the systems HARES is operating on are trusted.

Memory Protections. One weakness in the HARES prototype is its vulnerability to cold-RAM attacks and other memory-dumping techniques. In [2], a platform feature found on certain ARM system-on-chips is used to prevent sensitive data from leaving the SoC. Modern CPU caches have grown large enough to hold small programs, and integration and experimentation with such "no-fill" execution techniques are of particular interest as future work for HARES.

5. RELATED WORK

The following paragraphs aim to credit the previous works the HARES research builds on and to differentiate between these efforts and the work presented above. Keeping abreast of the research performed both in the academic and "hacker" spheres is an extremely challenging undertaking, as such this section should by no means be considered a complete survey of the field.

Packers. In essence, a packer is a generalized term for any run-time modification to a program, typically with the goal of avoiding signature-based checks of files prior to their execution. This term can encapsulate many sub-classes of program obfuscation techniques, such as VM-based malware, encryption tools, and compression. A packer works by creating a small program stub, executed at the start of a packed program's execution that "unpacks" the program from its on-disk format to that which is understandable by the host CPU environment. The program is unpacked into a memory region that is set to be executable, and execution is passed to this region after unpacking is complete.

A vulnerability of packers is their weakness to run-time reverse-engineering. While on disk the program's logic may be obfuscated or encrypted, once the unpacking routine is completed, the program will reside in memory, and be executed as a typical program, vulnerable to reverse-engineering and other semantic analysis.

BurnEye. An example of an encryption-protected packer, BurnEye was a protective method for GNU ELF files that allowed them to be encrypted and protected from static reverse-engineering, and seamlessly be decrypted and executed. Unfortunately, BurnEye is vulnerable to the same classes of attack as the more general packers: run-time and dynamic attacks allow the recovery of program instructions, rendering the protections insufficient for today's threat environment.

TrulyProtect. In [1], a CPU-bound VM interpreter is created that performs decryption on a randomized or encrypted intermediate representation (IR) while protected by a VMM. This work aims to accomplish similar goals, but requires a VM IR to x86 mapping, thus will not work on existing applications without access to the source. Additionally, there may be a risk of statistically reverse-engineering the encrypted mapping in a similar fashion to the risks associated with electronic codebook (ECB) mode with AES.

Sentry. In [2], researchers were able to attain a similar level of protections for binaries on ARM by leveraging platform-specific functionality, namely the AES_On_SoC bit to prevent protected memory regions from being written to any storage off the system-on-chip. The authors of HARES find the Sentry work incredibly valuable as mobile devices store an ever-increasing amount of sensitive information. An exciting area for future study would be an abstraction layer for encrypted software that can make use of the platform's capabilities: Intel SGX, ARM SoC locks and HARES for systems without such features.

6. CONCLUSION

In conclusion, HARES provides a significant improvement over the current state-of-the-art in protecting applications from reverse-engineering, though a determined adversary will certainly be able to defeat the protections. HARES provides a "drop-in" enhancement to the security posture for many applications known to be vulnerable, both against theft of algorithmic IP and denying a number of attack vectors. The upcoming Intel SGX capability will likely eclipse HARES for typical use, though the research presented herein is of interest nonetheless.

7. ACKNOWLEDGMENTS

The author would like to express his gratitude to the many parties who have helped on the HARES effort in one form or another, too many to mention all by name. Firstly, Mark Bridgman for his

help on the prototype development and debugging effort. Second, Loc Nguyen and Ryan Stortz for their input on weaknesses and areas of improvement. Finally, AIS for its support of this research and its publication.

8. REFERENCES

[1] A. Averbuch, M. Kiperberg, and N. Zaidenberg. Truly-protect: An efficient vm-based software protection. *Systems Journal, IEEE*, 7(3):455–466, Sept 2013.

[2] P. Colp, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 177–189, New York, NY, USA, 2015. ACM.

[3] B. Delgado and K. Karavanic. Performance implications of system management mode. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 163–173, Sept 2013.

[4] J. Gionta, W. Enck, and P. Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 325–336, New York, NY, USA, 2015. ACM.

[5] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. December 2009.

[6] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. In C. Kreibich and M. Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 21–40. Springer Berlin Heidelberg, 2010.

[7] T. Müller, F. C. Freiling, and A. Dewald. Tresor runs encryption securely outside ram. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 17–17, Berkeley, CA, USA, 2011. USENIX Association.

[8] L. Nguyen. Micronesia: Sub-kernel kit for host introspection in determining insider threat. ShmooCon, 2015.

[9] PaX Project. Pageexec, Mar 2003. `https://pax.grsecurity.net/docs/pageexec.txt`.

[10] J. Rutkowska. Red pill... or how to detect vmm using (almost) one cpu instruction, 2004.

[11] S. Sparks and J. Butler. Raising the bar for windows rootkit detection. *Phrack*, 17(61), November 2005.

[12] J. Torrey. More: Measurement of running executables. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, pages 117–120, New York, NY, USA, 2014. ACM.

[13] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures, 2009.

[14] P. Van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, pages 82–92, April 2005.

[15] R. Wojtczuk. Subverting the xen hypervisor. In *Black Hat USA*, Aug 2008.

[16] R. Wojtczuk and C. Kallenberg. Attacks on uefi security, inspired by darth venamis's misery and speed racer. CanSecWest, 2015.