Transparent System Introspection in Support of Analyzing Stealthy Malware

Ph.D. Dissertation Proposal

Kevin Leach
kjl2y@virginia.edu

May 12, 2015

1 Introduction

The proliferation of malware has increased dramatically and seriously damaged the privacy of users and the integrity of hosts in the past few years. Kaspersky Lab products detected over six billion threats against users and hosts in 2014 consisting of almost two million specific, unique malware samples [51]. McAfee reported that malware has greatly increased during 2014, with over 50 million new threats occurring during the fourth quarter [64] alone. While automated techniques exist for detecting pre-identified malware [85], manual analysis is still necessary to understand new and unknown threats. Malware analysis is thus critical to understanding new infection techniques and to maintaining a strong defense [16].

Malware analysis typically employs virtualization [25,26,34] and emulation [6,73,92] techiques that enable dynamic analysis of malware behavior. A given malware sample is run in a virtual machine (VM) or emulator. An analysis program (or a VM plugin) *introspects* the malicious process to help an analyst determine its behavior [17]. Introspection is a technique used to unveil semantic information about a program without access to source code. Initially, conventional wisdom held that the malware sample under test would be incapable of altering or subverting the debugger or VM environment [75,78]. Unfortunately, malware developers can easily escape or subvert these analysis mechanisms using several anti-debugging, anti-virtualization, and anti-emulation techniques [13,18, 23,33,39,72,74]. These techniques determine the presence of debuggers and VMs by using *artifacts* that they expose. We summarize known artifacts in Table 4 in the Appendix. Chen et al. [23] reported that 40% of 6,900 given samples were found to hide or reduce malicious behavior when run in a VM or with a debugger attached. Thus, the analysis and detection of such *stealthy* malware remains an open research problem which we call the *debugging transparency problem*.

There is a diverse array of anti-analysis artifacts. Artifacts come in two broad types: software-based and timing-based. Software artifacts are functional properties that can reveal the presence of analysis, such as the value returned by `isDebuggerPresent` or evidence of improperly emulated features. Timing artifacts are introduced by the cost of executing the analysis framework. Debuggers that allow single-stepping through instructions incur a significant amount of overhead, which can be measured by the software under test. Malware samples that are aware of these artifacts can thus heuristically detect when they are being analyzed. Such stealthy malware requires additional effort to understand. Solving the debugging transparency problem is thus concerned with reducing

artifacts or permitting reliable analysis in the presence of artifacts.

We choose to focus on three components to help analyze stealthy malware. Many analysts employ existing analysis tools such as IDA Pro [47] or OllyDbg [94], but such tools introduce slowdowns or other detectable artifacts. Thus, we desire a solution that provides *low overhead and low artifact* debugging capabilities. Second, these debugging capabilities ultimately rely on the ability to gather semantic information from a program. To provide information that the analysis requires, a solution should therefore successfully read and report 1) variable values and 2) dynamic stack traces [32]. Finally, black box testing of malware samples is an important step in understanding their behavior [16]. To promote black box testability, an expressive solution should also be able to insert instructions to specifically control behavior and write variable values.

To achieve these desired properties of a successful solution, we combine several insights to form the basis of a novel system for transparent malware analysis. First, specialized hardware exists that can be used to read a host's memory with extremely low overhead and without injecting artifacts in the platform's software. Second, we can use existing program analysis techniques to reconstruct valuable semantic information from raw memory dumps, including variable values and dynamic stack traces. Third, we can combine program analysis and specialized hardware to control program execution for low-artifact black box testing.

We propose a system consisting of three components that use the above insights to solve the debugging transparency problem. First, we propose low-artifact memory acquisition via novel use of the PCI-express bus as well as System Management Mode (SMM) on x86 platforms. PCI-express can access memory with low overhead (and therefore fewer timing artifacts), and with only one visible software artifact (the DMA access performance counter). SMM on x86 platforms can transparently acquire physical memory associated with a single process. Compared to the PCI express approach, using SMM introduces no software artifacts, but at the cost of increased overhead. Both approaches support a similar interface: the user provides an address, and the system returns the value of physical memory at that address.

Our second proposed component can use these snapshots of system memory and adapt well-known program analysis techniques to reconstruct semantic information about the program. If we can find the locations where variables are stored, we can determine their values from the memory snapshot. Similarly, since activation records of function calls are stored on the stack, we can reconstruct a sequence of function calls if we know where the program's stack is stored. We propose to use program analysis techniques to find this information given a binary and a raw memory dump.

Our third component allows modifying variables, especially those residing in registers. Whereas our introspection components may provide read-only access to variable values in a program, we also desire write access to variables and registers to permit effective black box testing. The insight here is the use of hardware support made for debugging firmware, which we propose to repurpose for inserting instructions. We propose to devise a technique whereby we construct sequences of instructions to accomplish certain behaviors associated with testing without causing significant overhead. This technique can be achieved using CPU interposition, which involves specialized hardware that sits between the platform's mainboard and CPU socket. The research problem at hand is whether we can use CPU interposition to transparently insert instructions into the CPU's execution units in a way that permits changing variables and controlling paths of execution taken by a program. While this component is preliminary, there is promise based upon experience with hardware debugger support in x86 and ARM architectures.

1.1 Broader Impact

Malware analysis is an increasingly important area. As malware becomes more complex, the strain on analysis resources escalates. Engineers in industry frequently take as long as one month to manually analyze a new single sample of stealthy malware [82], and many companies simply do not have the resources to analyze all stealthy infections [24]. With millions of new samples appearing every year, this time investment is not feasible, and is certainly no longer cost effective. Further, with the increased prevalence and reliance upon computing in our everyday lives, large-scale malware such as Stuxnet and Careto [53] have caused significant financial damage. As for individual users, Kaspersky Labs reported 22.9 million attacks involving financial malware that targeted 2.7 million individual users [52] in year 2014 alone.

1.2 Intellectual Merit

With the increased prevelance of stealthy malware, the debugging transparency problem has become an open and fertile area of research. Our proposed techniques are novel to the best of our knowledge. Solving this problem requires combining insights spanning programming languages, operating systems, hardware development, and security. First, our proposed memory acquisition approach requires strict compliance with the PCI express specification (to avoid software-based artifacts) while providing low overhead access (to avoid timing-based artifacts) to semantic information. Second, our proposed program analysis technique depends upon finding values of variables, which are in non-obvious locations (i.e., those chosen by an optimizing compiler). Third, we must automatically constrain desired testing behavior into a short sequence of inserted instructions while executing an adversarial program. No currently extant system shows promise in providing the benefits of binary program analysis in a transparent manner.

1.3 Thesis Statement

> It is possible to develop a transparent debugging system capable of reading and writing variable values, reconstructing stack traces, and black box testing by combining hardware assisted memory acquisition, process introspection, and CPU interposition.

This proposal consists of three research components that, together, form a cohesive system supporting the automated, transparent analysis of software. First, we discuss hardware-assisted system introspection techniques. Hardware assistance provides the necessary basis for analyzing software with low overhead, and therefore low artifacts. Second, we discuss transparent process introspection. This technique facilitates reading variable values and reconstructing dynamic stack traces by using said hardware assistance. Finally, we discuss CPU interposition. Whereas transparent process introspection admits reading variables in a process's memory, we propose to use CPU interposition to allow writing variable values in both memory and registers. Together, these techniques help solve the debugging transparency problem.

2 Background

In this section, we define terms and vocabulary used in this area.

2.1 Stealthy Malware

Recent malware detection and analysis tools rely on virtualization, emulation, and debuggers [13, 18, 22, 33] (also, see the Appendix). Unfortunately, these techniques are becoming less applicable with the growing interest in *stealthy malware* [23]. Malware is stealthy if it makes an effort to hide its true behavior. This stealth can emerge in several ways.

First, malware can simply remain inactive in the presence of an analysis tool. Such malware will use a series of platform-specific tests to determine if certain tools are in use. If no tools are found, then the malware executes its malicious payload.

Second, malware may abort its host's execution. For example, a sample may attempt to execute an esoteric instruction that is not properly emulated by the tool being used. In this case, attempting to emulate the instruction may lead to raising an unhandled exception, crashing the analysis program.

Third, malware may simply disable defenses or tools altogether. For instance, a previous version of OllyDbg would crash when attempting to emulate `printf` calls with a large number of '%s' tokens [93]. This type of malware may also infect kernel space and then disable defenses by abusing its elevated privilege level.

A firm understanding of stealthy malware is important due to its increasing prevalence and strain on analysis resources.

2.2 Artifacts

Stealthy malware evades detection by concluding whether an analysis tool is being used to watch its execution and then changing its behavior. This means there must be some piece of evidence available to the malware that it uses to make this determination.[1] This may be anything from execution time, (e.g., debuggers make programs run more slowly), to I/O device names (e.g., if a device has a name with 'VMWare' in it), to emulation idiosyncrasies (e.g., QEMU fails to set certain flags when executing obscure corner-case instructions). We coin a novel term for these bits of evidence: *artifacts*. Ultimately, we seek more transparent instrumentation and measurement of malware by reducing or eliminating the presence of these artifacts.

2.3 System Management Mode

Our first proposed research component makes extensive use of System Management Mode (SMM) [48]. SMM is a mode of execution similar to Real and Protected modes available on x86 platforms. It provides a transparent mechanism for implementing platform-specific system control functions such as power management. It is initialized by the BIOS. SMM's was originally designed to enable the hardware to save power by monitoring which peripherals were being used and turning off peripherals when idle. SMM is transparent to the operating system, so its use in power management did not require extensive driver support. We make use of this transparency aspect to execute analysis code unbeknownst to code executing in both user and kernel space.

[1] An analog in other scientific fields is the *observer effect*, which refers to observable changes that result from the act of observation.

SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be asserted in a variety of ways, which include writing to a hardware port or generating Message Signaled Interrupts with a PCI device. Next, the CPU saves its state to a special region of memory called System Management RAM (SMRAM). Then, it atomically executes the SMI handler stored in SMRAM. SMRAM cannot be addressed by the other modes of execution. This caveat therefore allows SMRAM to be used as secure storage. The SMI handler is loaded into SMRAM by the BIOS at boot time. The SMI handler has unrestricted access to the physical address space and can run privileged instructions. SMM is thus a convenient means of storing and executing OS-transparent analysis code.

2.4 Threat Model

The proposed system entails malware analysis. We must therefore define the scope of the malware that we propose to analyze. We assume a malware sample can compromise the operating system after executing its very first instruction. We further assume the malware can use unlimited computational resources. We assume that the physical hardware is trusted; hardware trojans are thus out of scope.

3 Related Work

In this section, we discuss the context in which our proposed work fits.

3.1 Malware Analysis

Stealth techniques employed by malware have necessitated the development of increasingly sophisticated techniques to analyze them. For benign or non-stealthy binaries, numerous debuggers exist (e.g., OllyDbg [94], IDA Pro [47], GNU Debugger [42]). However, these debuggers can be trivially detected in most cases (e.g., by using the `isDebuggerPresent()` function). Such anti-analysis techniques led researchers to develop more transparent, security-focused analysis frameworks using virtual machines (e.g. [6, 25, 26, 34, 80, 92]) which typically work by hooking system calls to provide an execution trace which can then be analyzed. System call interposition has its own inherent problems [38] (e.g., performance overhead) which led many researchers to decouple their analysis code even further from the execution path. Virtual-machine introspection (VMI) inspects the system's state without any direct interaction with the control flow of the program under test, thus mitigating much of the performance overhead. VMI has prevailed as the dominant low-artifact technique and has been used by numerous malware analysis systems (e.g. [28, 40, 44, 50, 59, 79, 83]). Jain et al. [49] provide an excellent overview of this area of research. However, introspection techniques have very limited access to the semantic meaning of the data that they are acquiring. This limitation is known as the *semantic gap problem*. There is significant work in the semantic gap problem as it relates to memory [11, 29, 36, 50, 60] and disk [20, 55, 62] accesses. All VM-based techniques thus far have nevertheless been shown to reveal some detectable artifacts [23, 72, 74, 76] that could be used to subvert analysis [35, 71].

The semantic gap problem requires reconstructing useful information from raw data, but this reconstruction process is completely separate from the method of acquisition. Numerous techniques

have been discussed which further decouple the analysis code from the software under test by moving the analysis portion into System Management Mode (e.g. [12,86,87,95]) or onto a separate processor altogether (e.g. [15, 65, 66, 69, 96]). In contrast to existing work, we consider SMM as a trusted mechanism for transparently acquiring physical memory associated with a particular process.

3.2 Debugging

Most popular debuggers, such as the GNU Debugger (GDB), Pin [61], OllyDbg [94], or DynamoRio [19], work by modifying the instructions of the process being debugging by adding jump or interrupt instructions that will periodically call back to the debugging framework to report the requested data. These techniques have obvious performance impacts as they execute multiple instructions in-line within the debugged process, and are known to be trivial to detect. More security-focused dynamic analysis engines either emulate the CPU in software [14,43,90] or execute the instructions on the hardware and monitor memory accesses or system calls to trigger their instrumentation [27, 79, 81, 95]. However, these techniques have also been shown to expose numerous artifacts and be susceptible to subversion [35, 71].

Hardware debuggers, such as JTag, Asset InterTech's PCT [9], or ARM's DStream [8], expose no software artifacts, but still have a significant performance impact as they typically function by iteratively sending a set of instructions to the CPU and analyzing system state upon completion of those instructions. These debugging interfaces are typically orders of magnitude slower than the CPU that they are debugging, and thus introduce inherent performance restrictions when used as a general debugging platform. The aforementioned timing constraints make single-stepping techniques cumbersome for large programs, and are also easily detectable by malicious programs. Further, non-malicious processes dependent on time such as network servers may be unstable in such environments. This instability could be treated as a separate artifact that malicious code could use in turn. These techniques have also been widely unexplored in a security context and are typically reserved for debugging the Basic Input/Output System (BIOS) and boot loaders. This approach explores low-artifact hardware-based debugging techniques for general software programs. Furthermore, we revisit the use of these hardware debuggers as a mechanism for inserting single instructions rather than for general debugging.

3.3 Memory Introspection

The idea of memory introspection is by no means a new field of study. Numerous tools have been created to facilitate memory introspection on both physical and virtual machines over the years [21,30,63,68,87], as well tools that bridge the semantic gap. Bhushan et al. [49] summarized the research in the field with virtual machines, however the area of live hardware-based memory introspection is mostly unexplored. In general, these techniques require instrumentation to acquire memory, which is augmented to bridge the semantic gap to extract high-level information from the system being analyzed. Many analysis techniques use expert knowledge to reconstruct features of popular operating systems and processes, as in Volatility [11]. Others use automated techniques to reproduce native process functionality, such as enumerating running processes, via an introspection framework [29, 36, 50, 60]. A third class of techniques achieve isolation by moving security critical introspection functions to a higher level hypervisor or external process [41, 83]. While a few hardware-based techniques currently exist [69, 87, 96], they either have a very specific focus

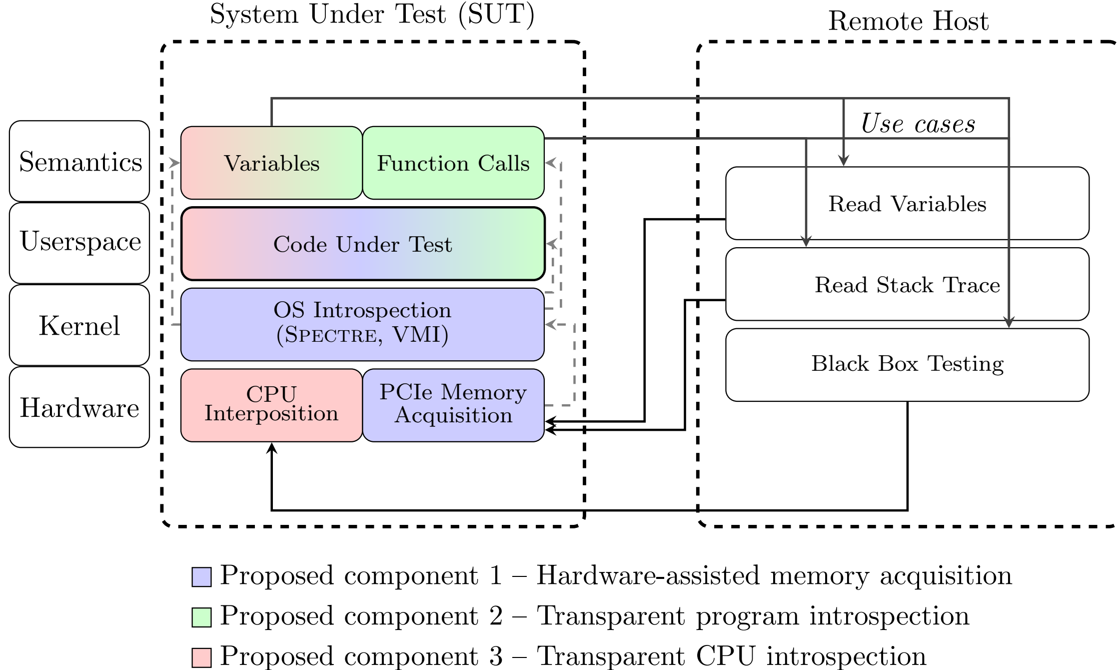


Figure 1: Proposed system structure.

or a significant performance overhead. We are unaware of any hardware-based systems capable of analyzing arbitrary programs and providing debugging-like information that includes variable values and stack traces while maintaining transparency from the software under test.

4 Proposed Research and Metrics

We propose a system consisting of two hosts, 1) a *system under test* (SUT), and 2) a *remote host*. The SUT is the platform that contains the code we want to observe (e.g., a stealthy malware sample), while the remote host is used by an analyst to guide introspection and debugging on the SUT. This architecture is favorable because it isolates duties: the SUT executes its code under test transparently. The remote host, in turn, is responsible for debugging and analysis. Figure 1 gives a high-level architecture diagram of our system; in this section we discuss each proposed research activity in detail.

4.1 Hardware-Assisted Program Introspection

To support and carry out transparent malware analysis we require transparent access to a host's memory. We propose two techniques to achieve this end. First, we propose a novel use of the PCI express bus to rapidly acquire memory contents via direct memory access (DMA). This technique has the benefit of producing very little measurable overhead and minimal software artifacts—specifically, software can measure the DMA performance counter. We also propose a novel use of System Management Mode (SMM) to acquire process memory transparently. SMM potentially produces timing artifacts, but it does not produce any software artifacts. Both techniques provide a

similar interface for our purposes: a desired address is given as input, and the value of the process's memory at that address is returned as output.

4.1.1 Using PCI express

We propose using a Field-Programmable Gate Array (FPGA) board with a PCI express connector to create a *sensor* capable of acquiring memory values transparently. PCI express supports very high speed transfers. Customized hardware in the form of an FPGA allows us to take full advantage of this high bandwidth. Thus, we can acquire many samples of memory values with low overhead, and therefore few timing artifacts. This approach is also attractive because we can use commercial off-the-shelf (COTS) components to develop the sensor. As a result, we can potentially drastically lower the effort involved in acquiring memory values.

Specifically, we propose to use a Xilinx ML507 development board. This FPGA has PCI express and gigabit ethernet connectors. We propose designing an FPGA that supports reading memory over PCI express via DMA, then passes the values over Ethernet to a remote host. In our proposed setup, the board communicates with the system under test via PCI express. We achieve this by enabling the *Bus Master Enable* bit in the card's configuration register. As hardware is typically trusted, there are no enforcement mechanisms to stop a peripheral from reading arbitrary memory locations. This method has been widely studied [21, 77, 87] and exploited [10, 30, 31, 45, 56, 77, 84]. However, this caveats use in transparent memory acquisition for the legitimate study of stealthy malware is novel. A remote host connects via gigabit Ethernet. An analyst then uses the remote host to orchestrate an introspection session on the system under test.

4.1.2 Using System Management Mode

To use SMM to acquire memory values, we logically atomically execute the System Management Interrupt (SMI) handler by asserting the SMI pin on the CPU. On most Intel platforms, we assert the SMI pin by writing to a port specified by the chipset. By modifying the SMI handler to contain process introspection code, we can provide simple debugging services to an external host.

The SMI handler executes in isolation from the operating system, thus providing transparency. Moreover, the SMI handler code is stored in the BIOS on the system. Upon booting, the SMI handler is loaded from the BIOS EEPROM to a special region of memory called System Management RAM (SMRAM). SMRAM is only addressable when the CPU is executing in SMM. Otherwise, the hardware automatically maps SMRAM addresses to VGA memory instead. As a result, SMRAM can be used as trusted storage because its isolation is enforced by the hardware, which we assume is trusted. SMM thus provides a transparent execution environment that permits us to acquire memory values even if the underlying operating system has been compromised.

We propose using SMM to provide a memory acquisition interface. In the system described, we must assert interrupts to cause our introspection code to execute. Typical computer chipsets support raising an SMI when a performance counter overflows. Therefore, to trigger an SMI in the immediate future, we propose to manually set the value of a performance counter to its maximum value. When the corresponding event next occurs, the performance counter overflows, raising an SMI. If needed, we can use the retired instruction performance counter to achieve an SMI after every single instruction executed by the CPU. This allows us to account for local timing

artifacts by altering hardware counters on every instruction, thus preventing software artifacts from occurring.

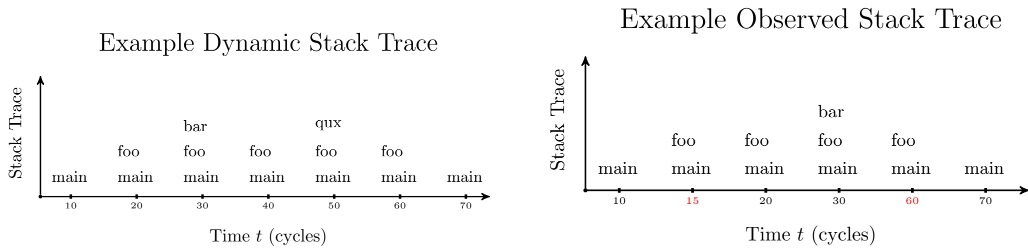
4.2 Transparent Program Memory Introspection

The second component of our proposed transparent malware analysis system constructs useful semantic information from raw memory dumps. In particular, we desire to introspect variable values and runtime stack information from memory dumps acquired from our proposed hardware-assisted memory acquisition component described in Section 4.1. However, these raw physical memory snapshots do not come with meaningful semantic information—we must bridge the semantic gap. Thus, we propose a technique that iteratively inspects snapshots of physical memory (recorded asynchronously via the first proposed component) and recovers semantic information. Using a combination of operating systems and programming languages techniques, we locate local, global, and stack-allocated variables and their values. In addition, we determine the current call stack. Because we may operate on off-the-shelf optimized code and only assume access to memory (e.g., and not the CPU), our approach may not be able to report the values of some variables (such as those stored in registers) or some stack frames (such as those associated with inlined functions). We believe this approach will be able to provide a rich set of information for common security analysis tasks in practice.

We envision two use cases for this proposed component. In automated malware triage systems, we want to analyze a large corpus of malware samples as quickly as possible. Unfortunately, existing solutions to this problem depend on virtualization. For example, the popular Anubis [6] framework, which analyzes binaries for malicious behavior, depends on Xen for virtualization. This dependency on virtualization allows stealthy or VM-aware malware to subvert the triage system. In contrast, this proposed component assumes the presence of low-overhead sampling hardware that enables fast access to a host's memory. This proposed component, therefore, is charged with introspecting the process while running in the triage system. For such a triage system, we want to understand a subject program's behavior in part by reasoning about the values of variables and producing a dynamic stack trace during execution. For example, the system might inspect control variables and function calls to determine how a stealthy malware sample detects virtualization, or might inspect snapshots of critical but short-lived buffers for in-memory cryptographic keys. In this use case, the primary metric of concern is accuracy with respect to variable values and stack trace information.

Additionally, we envision this use case as reducing the manual effort involved in reverse engineering state-of-the-art stealthy malware samples. This proposed component enables debugging-like capabilities that are transparent to the sample being analyzed. This power allows analysts to save time when reverse engineering the anti-VM and anti-debugging features employed by current malware so that they can focus on understanding the payload's behavior.

The second use case is for deployments in large enterprises where multiple instances of the same software are running on multiple hosts. In this case, we have vulnerable software running and, if an exploit occurs, we want to know what memory locations are implicated in the exploit. Here, we have a limited amount of time between when malicious data is placed into program memory and when malicious behavior begins. Thus, we want to study which buffers may be implicated by malicious exploits in commercial off-the-shelf software. In this use case, an additional metric is the relationship between the speed and accuracy of our asynchronous debugging approach: for



(a) Example ground truth dynamic stack trace. The dynamic stack trace is a time series of static call stacks showing which functions are called over time in the program.

(b) This example set of stack trace observations demonstrates the tradeoffs between accuracy and transparency. Sampling too frequently may require additional resources or introduce artifacts without yielding additional information (e.g., sampling at $t = 15$ and $t = 20$). Conversely, sampling too coarsely may cause miss function calls (e.g., between $t = 30$ and $t = 60$).

Figure 2: Visual representation of (a) an idealized stack trace and (b) a stack trace generated by the proposed component.

example, we may require that accurate information about potentially-malicious data be available quickly enough to admit classification by an anomaly intrusion detection system.

In both use cases, we begin with a binary that we want to study. If the source code to this binary is available (as is likely in the second use case but unlikely in the first), we take advantage of it to formulate hypotheses about variable locations. In any case, we desire to find and report 1) variables of interest in program memory, and 2) a dynamic stack trace of activation records to help understand the semantics of the program.

We assume that we have a binary compiled with unknown optimizations or flags. We also assume access to the source code for the binary. We compile a test binary with symbol information to formulate the runtime memory locations of global and local variables and formal parameters. We then use this information as a guide for finding the same variables in the original binary. For instance, if global a is stored at offset x from the data segment in the test binary, then we hypothesize a is near the same offset x in the original binary.

We also assume the binary makes use of a standard stack-based calling convention (i.e., continuation passing style [7] is out of scope). Thus, we can reconstruct a stack trace by finding the stack in our memory snapshots and looking for the return address. We can then map the return address to the nearest enclosing entry point in the code segment using the symbol table generated in our test binary. Because the hardware assisted introspection component is based on polling for memory, reconstructing the stack trace in this manner could fail. If we do not poll frequently enough, we may miss activations altogether (e.g., if a function is called and quickly returns before we can poll). Figure 2 explains this tradeoff visually.

4.3 Transparent CPU Interposition

The third proposed component is more speculative, but holds promise nonetheless. As such, we propose two optional components as a risk mitigation strategy to ensure successful completion

of the overall proposed research. After sufficient preliminary investigation is completed, we will select one of the potential components to take to completion. Both potential components relate to CPU interposition as part of completing the overall solution to the debugging transparency problem.

4.3.1 Option 1: Black Box Testing of Android Malware

To address the growing concern of mobile malware [64] and stealthy Android malware [70], we propose a CPU interposition component to apply to ARM platforms, specifically Android. In particular, the process introspection component introduced in Section 4.2 permits reading of variables and stack traces. While this can help analysts understand a sample's behavior, it may also be useful to change a sample's behavior. For example, we claim that Android malware targeting financial information from users may only activate if a user's bank account has a certain balance. In this example, the balance would be stored in a variable. Thus, we would need to change this variable to observe changes in the execution of the malware sample. Moreover, general debugging tools such as GDB [42] or IDAPro [47] allow changing variable values when interacting with a piece of software. We propose a CPU interposition component so that our system offers a complete set of debugging features while remaining transparent. We thus propose to allow the transparent alteration of such a variable to explore different execution paths.

We propose using hardware debuggers available for ARM platforms (e.g., the ASSET PCT for ARM [9] or the ARM DSTREAM debugger [8]). This debugging hardware is intended to test manufacturing defects and to test low level firmware. However, based upon our preliminary investigation, these tools can be extended to help transparently analyze stealthy malware. In particular, these hardware debuggers function via interposing the CPU–that is, a special circuit exists between the CPU socket on the platform's mainboard and the CPU itself. The hardware debugger has access to the CPU's pins and therefore can intercept all signals sent between the CPU and the rest of the system.

Hardware debuggers allow two general operations: 1) reading and writing arbitrary memory values in registers, caches, and DRAM banks, and 2) inserting instructions by shifting a new instruction into a special register that feeds to an execution port in the CPU pipeline. Further, hardware debuggers can be invoked at regular intervals (e.g., 20MHz for JTAG-based debuggers). Therefore, by using the program introspection component from section 4.2, we can identify locations of variables in memory, and change them using CPU interposition. Furthermore, we can insert instructions to change program behavior in a way that is difficult for stealthy malware to detect. Together, these powers allow us to complete our proposed solution to the debugging transparency problem.

We propose an approach whereby we alter the outcome of branches in software under test by changing the value of a relevant variable. With this proposed approach, we assume a malware analyst indicates the desired path to be exercised by a particular branch by asserting the value of a variable. Ultimately, this component will be successful if we can change the value of the variable to effect a change in the outcome of the branch. The challenge is thus whether we can change the value without influencing overall performance (i.e., without measurable slowdown) and whether we can change the value in time to make a difference (e.g., if a variable's value is changed after the branch is taken, then the component fails).

4.3.2 Option 2: Realtime Repair of Quadcopter Systems

Realtime software provides another compelling need for reducing timing artifacts. Expensive software analysis techniques may take too long to complete or fail altogether in the context of a realtime or resource-constrained environment. One such area of growing interest is in unmanned aerial vehicles. In particular, we want to increase the dependability of software running on quadcopters despite a diverse heterogeneity of specific hardware and firmware configurations that lead to failures [37]. Such realtime software would benefit from techniques that fix bugs on the fly to prevent mission failures.

Automated program repair is an active area of research concerned with automatically generating patches to fix bugs in software [57, 88, 89]. Typically, these techniques assume the existence of a comprehensive suite of test cases that help localize bugs in the software. The original software passes some fraction of the test cases. Then, by changing the code localized by the automated repair system, a patch can potentially be generated that causes the software to pass a larger fraction of the test cases. However, automated program repair has not been used on firmware, since it requires changing the program code while it is stored (as opposed to changing the code at runtime). While we do not propose new automated program repair techniques, we seek to apply these techniques in a novel domain.

We propose a component in which we apply automated program repair by using CPU interposition. As mentioned in option 1, we can repurpose hardware debuggers to help repair realtime software. Specifically, debugger hardware permits *inserting* instructions only–it cannot remove instructions from the execution pipeline. Thus, we propose using ARM hardware debuggers to apply program repairs by inserting instructions only. A successful approach for this component requires automatically converting a general software patch into a sequence of instructions to be inserted and then reliably inserting them with minimal overhead. For example, if our patch requires removing an instruction `add rax, rax, 5`, we can simulate removing this instruction by inserting a subsequent `sub rax, rax, 5` immediately (ideally) after the first instruction. Thus, this component is charged with 1) coming up with a sequence of instructions to insert that represent the behavior of the desired patch, and 2) reliably inserting the patch in the desired location in a way that the new patch will execute.

We assume that a patch exists provided as a source code delta for a function or method to be changed. We also assume that we are provided the location for such a patch to be placed in the binary (i.e., the value of the program counter at which the patch should be placed). Therefore, we must automatically convert the provided patch to a sequence of instructions to be inserted via the hardware debugger *as soon as* the program counter reaches the desired value. This use case is indicative of self-checking malware–we must change the program code before it has a chance to execute, but not so early that it can check itself for discrepancies. Thus, patching quadcopter firmware is indicative of the same technical challenges posed by analyzing self-checking stealthy malware samples.

5 Proposed Experiments

In this section we present and discuss the metrics for evaluating the success of each of our three proposed components. We then present experimental designs for testing our hypotheses.

5.1 Hardware-Assisted Program Introspection

This subsection deals with the proposed hardware-assisted introspection technique. Recall that the desired properties of an adequate introspection technique are that it transparently acquires physical memory from a system under test. In that regard, we have two basic metrics for transparency: 1) latency of operation, and 2) freedom from software artifacts as measured by success in analyzing stealthy samples. The latency of the system helps indicate the degree to which it is free from timing artifacts. Together, these two metrics give an overall picture of how transparent the system is.

We will evaluate two hardware-assisted introspection techniques based on 1) System Management Mode (SMM) and 2) PCI express based DMA.

5.1.1 SMM-Based Introspection

We measure the latency of the SMM-based introspection approach by measuring the time elapsed during particular operations in the SMI handler. We use the Time Stamp Counter (TSC) to measure the number of CPU cycles elapsed during each operation; we multiply the clock frequency by the delta in the TSC. In particular, we measure the latency of four steps: 1) SMM switch – the time taken to switch to SMM, 2) Process identification and reading, 3) Configuration for the next SMI, and 4) SMM resume – the time taken to switch from SMM back to the OS.

We also measure the overall performance impact on a test system by comparing the run time of several programs in two situations: 1) when the system under test is *not* instrumented by our proposed component, and 2) when our system under test is instrumented on each instruction. To this end, a successful solution will be able to analyze more stealthy malware samples than existing techniques such as Ether [26] or Virtuoso [58] while incurring a lower performance overhead.

To measure freedom from software artifacts, we instrument popular stealthy malware samples with our proposed components and compare the results against other state of the art malware analysis techniques. We hypothesize that our technique will be able to instrument execution of more samples than other techniques such as Ether or Virtuoso.

5.1.2 PCI Express Hardware Introspection

We measure overall memory throughput for our proposed component. We compare the throughput on the system under test when it is 1) uninstrumented and 2) instrumented by our component. Specifically, we measure the number of MB per second of transactions that we can complete using a popular tool, RAMSpeed [46]. When the instrumented memory throughput statistically matches the uninstrumented memory throughput, we consider the component transparent (i.e., free from timing artifacts) and thus successful.

We then measure its effectiveness in analyzing stealthy malware samples. We consider a corpus of labeled malware samples that serve as ground truth for analysis. Within this labeled corpus, we know which of them contain stealthy checks for artifacts. Thus, we count how many of the stealthy samples this component is capable of analyzing. If we successfully analyze all of them, the component is successful.

5.2 Transparent Process Introspection

There are three parts we consider for evaluating the transparent process introspection component. First, we want to know what fraction of local, global, and stack-allocated variable values our component can successfully and correctly report. Second, we measure how accurately this component can recover dynamic stack traces from memory snapshots as a function of sampling frequency. Finally, we apply the component to a case study to show its usefulness in analyzing stealthy malware samples.

To address the first part, we assume two binaries for each program we consider: 1) the original binary, and 2) a test binary. We gather ground truth data by recording every variables value that is in-scope and available at every point of execution in the test binary by using source-level instrumentation [67]. Then, we execute the original binary. At each time step for which our component records a snapshot, we report the value of the variable. We measure success in terms of the fraction of those variables for which our component reports the correct value as reported in the test binary.

For the second part of this evaluation, we again assume we have two binaries for each subject. We gather ground truth information about function calls during program execution. We then vary how frequently we can acquire memory snapshots while running the original binary. From these snapshots, we find the activation stack and acquire the current function being executed at each snapshot. We then compare the stack trace acquired with memory snapshots to the ground truth. Thus, we evaluate the portion of the dynamic stack trace that our component can accurately report given a memory snapshot every k cycles. We consider function calls only in userspace (e.g., program functions like `main` and library functions like `printf` are included, but not actions like system calls).

We introduce a new metric for this experiment—the number of function calls missed in our output stack trace that were present in the ground truth stack trace. For example, if the ground truth sample contains $\langle(1, f_1), (2, f_1 \rightarrow f_2), (3, f_1 \rightarrow f_2 \rightarrow f_3)\rangle$ and we sample at cycles $1, 3, 5, \ldots$, then our approach would report a stack trace missing the call to f_2 at $t = 2$. Our metric counts the total number of such single omissions. Because the stack trace length differs among test cases and programs, we normalize this value to 100%. Thus, if a stack trace identical to the ground truth corresponds to an accuracy of 100% while the example above that misses program behavior at $t = 2$ has an accuracy of 66%. In other words, the final value we report is $\frac{f-m}{f}$, where m is the number of misses and f is the number of function calls in the ground truth data. While other evaluation metrics are possible for dynamic stack traces (e.g., longest common subsequence, edit distance) we prefer this metric because of its conservative nature.

In addition, for the third part of this evaluation we consider *pafish* v04. Pafish is an open source program for Windows that uses a variety of common artifacts to determine the presence of a debugger or VM, printing out the results of each check. This evaluation requires that we know the ground truth set of artifacts that pafish considers to detect analysis. We obtain this set by manual inspection of the pafish code and comments. In particular, pafish runs through a list of known artifacts and reports whether or not each artifact is detected during execution. Using pafish is amenable to complete ground truth annotation (unlike a wild malware sample for which we could miss a stealthy check and thus have false negatives). Further, a case study of pafish gives confidence that the proposed component is applicable to the analysis of stealthy malware because it contains a large set of indicative artifacts used by samples in the wild.

5.3 CPU Interposition

We proposed two options for this component in Section 4.3. As such, we propose experiments to that help fail early so as to inform our decision about which option to take to completion.

In Section 4.2, we proposed a memory introspection technique for x86 platforms. Since the first proposed option for CPU interposition is planned for ARM platforms, a fast experiment is whether we can quickly retarget the process introspection component on an Android platform. If we can, then we gain confidence that the first option is more appropriate.

Secondly, the proposed option for quadcopter repairs is based on automated program repair. Thus, if we can quickly retarget an existing automated repair system such as GenProg [57] to the ARM platform, then we gain confidence that the second option is more appropriate.

Additionally, we also propose translating patches to an equivalent sequence of inserted instructions for option 2. We propose manually making such translations for a candidate set of indicative patches. We then consider the types of operations required to achieve this behavior. We hypothesize that patches will involve a reasonably low number of common operations. For example, a large number of patches may exhibit the need to change behavior by inserting an instruction that 'reverses' the effect of a previous instruction. If patches can be classified into fewer than ten groups, we gain confidence that automating the translation is feasible, and thus that option two is viable.

After selecting an option, we consider further experiments.

5.3.1 Option 1: Android Black Box Testing

Recall from Section 4.3.1 that we want to modify the branch exercised by a sample of software. We assume we have the location of the variable to be changed (based upon analysis provided by the process introspection component in Section 4.2). We will evaluate every branch in a program. Each branch instruction is associated with a value in a register at that point of execution. We would attempt to change the value of that register so that the path taken by the branch is changed. If the fraction of branches whose results are changed in time due to our component exceeds 80%, we succeed.

5.3.2 Option 2: Repairing Quadcopter Firmware

We propose translating known patches into an equivalent representation of instructions to be inserted only (as required by our proposed use of hardware debuggers). Thus, we propose measuring the fraction of patches that we can successfully translate to such a representation. Then, we propose inserting these patches into a quadcopter as it executes its firmware. In particular, we use a hardware debugger to periodically check the program counter, compare it against the desired location of insertion, and determine whether to insert the patching instructions. For each patch, we measure the fraction of patches that can be successfully inserted. In particular, we must ensure that the patch is inserted at the desired location *before* execution reaches that location. We consider this component successful when it is capable of inserting a patch at least 80% of the time.

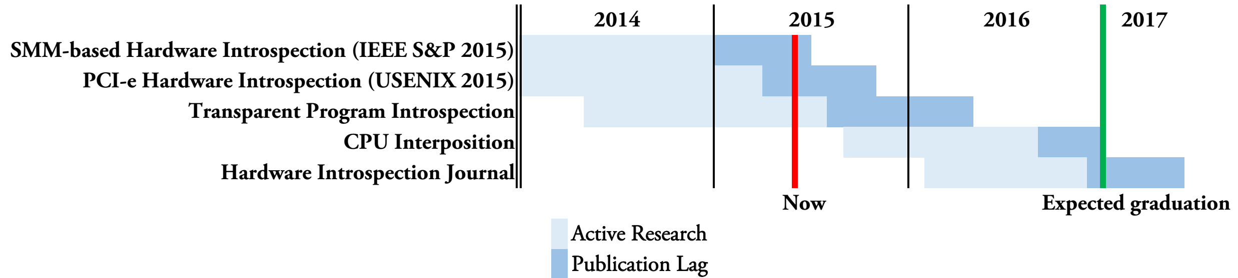


Figure 3: High level schedule for proposed research

6 Research Timeline

Figure 3 shows the overall plan from start to graduation of the proposed doctoral research. We will mainly target security venues for submitting original manuscripts. In particular, the highly reputable conferences and symposia in this area are 1) IEEE Security and Privacy, 2) USENIX Security Symposium, 3) Networks and Distributed Systems Security, and 4) ACM Conference on Computer Security. Additionally, other international conferences include 5) Annual Computer Security Applications Conference, 6) Recent Advances on Intrusion Detection, and 7) ACM Symposium on Information, Computer, and Communications Security. The submission dates for these annual conferences and symposia are roughly spread throughout the year, so there is always a nearby submission date to consider as research concludes on a particular research thrust.

7 Preliminary Results

In this section, we present preliminary results from ongoing research.

7.1 Hardware-Assisted Introspection

7.1.1 Using System Management Mode

Table 1 shows a breakdown of the operations in the SMI handler if the last running process is the target malware in the instruction-by-instruction stepping mode. This experiment shows the mean, standard deviation, and 95% confidence interval of 25 runs. The SMM switching time takes about 3.29 microseconds. Command checking and breakpoint checking take about 2.19 microseconds in total. Configuring performance monitoring registers and SMI status registers for subsequent SMI generation takes about 1.66 microseconds. Lastly, SMM resume takes 4.58 microseconds. Thus, the SMM approach takes about 12 microseconds to execute an instruction without debugging command communication. In contrast, the similar state of the art system Virtuoso [58] requires, in the best case, 450ms to complete a similar operation in Windows. Thus, this approach shows promise in providing improved timing transparency compared to existing work.

To demonstrate the efficiency of this component, we propose measuring the performance overhead of the four stepping methods on both Windows and Linux platforms. We propose using popular benchmark programs on each platform. We can also take advantage of CygWin and evaluate

Table 1: Breakdown of SMI Handler (Time: μs)

Operations	**Mean**	**STD**	**95% CI**
SMM switching	3.29	0.08	[3.27,3.32]
Program acquisition	2.19	0.09	[2.15,2.22]
Next SMI configuration	1.66	0.06	[1.64,1.69]
SMM resume	4.58	0.10	[4.55,4.61]
Total	11.72		

Table 2: Stepping overhead on Windows and Linux (Unit: times slowdown)

Stepping Methods	**Windows**					**Linux**				
	π	*ls*	*ps*	*pwd*	*tar*	π	*ls*	*ps*	*pwd*	*tar*
Retired far control transfers	2	2	2	3	2	2	3	2	2	2
Retired near returns	30	21	22	28	29	26	41	28	10	15
Retired taken branches	565	476	527	384	245	192	595	483	134	159
Retired instructions	973	880	897	859	704	349	699	515	201	232

performance of several standard Linux utilities. We can then use shell scripts to compute the runtimes of these commands.

Table 2 shows the performance slowdown introduced by the step-by-step debugging for several programs. The first column specifies four different stepping methods; the following five columns show the slowdown on Windows, which is calculated by dividing the current running time by the base running time; and the last five columns show the slowdown on Linux. Far control transfer (e.g., call instruction) stepping only introduces a 2x slowdown on Windows and Linux, which facilitates coarse-grained tracing for malware debugging. As expected, fine-grained stepping methods introduce more overhead. Preliminary evaluation of instruction-by-instruction debugging causes about 973x slowdown on Windows in the worst case. This high runtime overhead is due to the 12-microsecond cost of every instruction (as shown in Table 1) in the instruction-stepping mode. While this overhead is high, our proposed approach is still three times faster than previous work such as Ether [26, 92] in single-stepping mode.

7.1.2 Using PCI Express and DMA

We quantify our performance impact by utilizing *RAMSpeed*, a popular RAM benchmarking application. We run RAMSpeed with and without our instrumentation. In each case we ran four separate experiments which stressed the INT, MMX, SSE, and FL-POINT instruction sets. Each of these experiments consists of 500 sub-experiments which evaluate and average the performance of copy, scale, sum, and triad operations. To ensure that the memory reads for our instrumentation were not being cached, we had our sensors continuously read the entire memory space which should also introduce the largest performance impact on the system. The memory polling rates of 14MB/s was dictated by the available hardware and our particular implementation.

Figure 4 shows the overhead incurred by our PCI-e memory acquisition component as reported by RAMSpeed. At first glance, it may seem to indicate that our instrumentation has a discernible effect on the system. Note, however, that the deviation from the uninstrumented median is only 0.4% in the worst case (the SSE boxes in Figure 4). Despite our best efforts to create a controlled experiment, i.e. running RAMSpeed on a fresh install of Windows with no other processes, we were unable to to definitively attribute any deviation to our polling of memory. While our memory

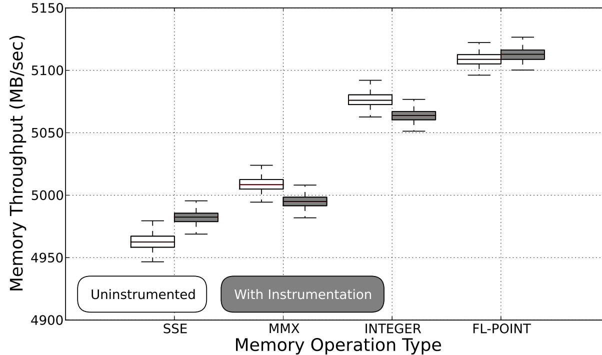


Figure 4: Average memory throughput comparison as reported by RAMSpeed, with and without instrumentation (500 samples per box).

Table 3: Variable introspection accuracy.

	nullhttpd		**wuftpd**	
Locals	43%	133 / 306	46%	202 / 436
Stack	65%	168 / 260	56%	119 / 214
Globals	100%	77 / 77	92%	4218 / 4580
Overall	59%	378 / 643	90%	4539 / 5230

polling certainly has some impact on the system, the rates at which we are polling memory do not predictably degrade performance. These results indicate that such a system could poll at significantly higher rates while still remaining undetectable. For example, PCIe implementations can achieve DMA read speeds of 3GB/sec [5] which could permit a novel class of introspection capabilities.

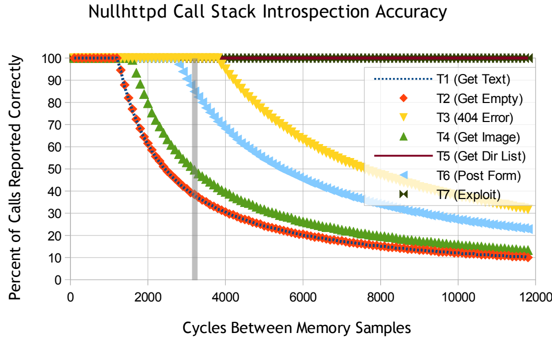
7.2 Transparent Program Introspection

We evaluate this proposed component using two indicative security-critical systems, *nullhttpd* 0.5.0 and *wu-ftpd* 2.6.0, each of which has an associated security vulnerability and publicly-available exploit. Nullhttpd is vulnerable to a remote heap-based buffer overflow [2] while wu-ftpd is vulnerable to a format string vulnerability [1]. We consider indicative non-malicious test cases taken from previous research [57].

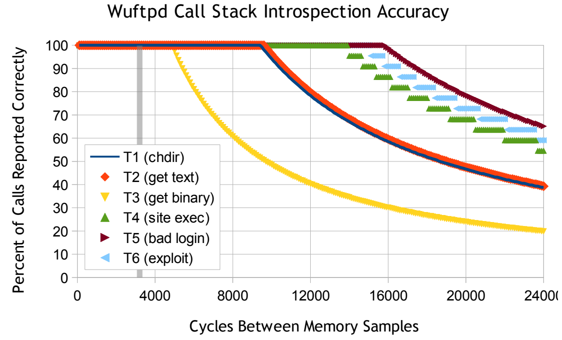
Table 5 in the appendix summarizes the test cases used in our experiments. When the source code is available, our approach uses it to construct hypotheses about variable locations, but we do not assume that the compiler flags used in the original binary are known or the same. In these experiments, we simulate that disparity by gathering hypotheses from programs compiled with "–O2," but evaluating against different binaries produced with "–O3".

7.2.1 Variable Reporting Accuracy

We evaluate the accuracy of our introspection component with respect to polled memory snapshots as discussed in Section 5.2. We partition variables in scope in to local, global, and stack-allocated



(a) Call stack introspection accuracy for Nullhttpd as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 1200 cycles yields perfect accuracy.



(b) Call stack introspection accuracy for Wuftpd as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 4800 cycles yields perfect accuracy.

variables. Table 3 reports the results of our variable introspection experiment. For each program, the results are averaged over all test inputs (non-malicious indicative tests and one malicious exploit) and all relevant points (all function calls, all function entries, all loop heads). That is, these results help address the question "if an analyst were to ask at a random point to introspect the value of a random variable, what fraction of such queries would our system be able to answer correctly?"

These results show over 90% accuracy for global variables. This high introspection accuracy is because many of these variables are available at a constant location described in the subject program's symbol table. However, our approach still produces reasonable introspection accuracy for local and stack-allocated variable queries. For these variables, the values are not necessarily unavailable, but the hypotheses considered by our system do not account for differences caused by dynamic allocation of structures (or, indeed, whether compiler optimizations change the structures or layout altogether). Conversely, some values are not available to our technique based on its design assumptions (e.g., variables that live exclusively in registers). Overall, however, this component answered 84% (4917 of 5873) of variable introspection queries correctly.

Figure 5a reports the results for stack trace introspection for nullhttpd. Current PCI-e DMA hardware can read roughly 1 million pages per second, or 1 page every 3200 cycles. Thus, current hardware approximately corresponds to 3200 cycles between memory snapshots in these figures. Our introspection system loses accuracy when functions execute faster than the chosen inter-sample cycle count. Each test case causes a different execution path to be taken, thus explaining the difference in results between test cases. For nullhttpd, we remain 100% accurate until the inter-sample cycle counter reaches approximately 1800 cycles. After this point, the accuracy steadily declines until the inter-sample cycle count exceeds the total execution time of the program — at that point, the accuracy is 0%. Note that with the 3200 cycle sample rate, we observe a stack trace accuracy over 50% for all test cases.

Figure 5b reports the accuracy for stack trace introspection for wuftpd. This program, which contains longer-running procedures, admits perfect call stack introspection up to a sampling interval of 4800. With available PCI-e DMA hardware, this component would report 100% accurate stack traces for all test cases. For such programs and workloads, a faster sampling rate (i.e., a smaller inter-cycle rate) may allow for even greater introspection transparency.

8 Conclusion

The rapid proliferation of malware has recently increased dramatically, seriously damaging the privacy of users and the integrity of hosts. With the sizeable growth of stealthy malware, novel techniques must be developed that remain transparent to the sample being analyzed. We propose a multi-faceted system comprised of three components: hardware-assisted program introspection, transparent process introspection, and CPU interposition. We propose using hardware assistance to rapidly acquire values of memory addresses from a host without introducing functional or timing artifacts. Then, we propose using this access to memory to transparently reconstruct the semantics programs, specifically to acquire variables and stack traces. Finally, we propose two options for demonstrating CPU interposition as a viable means to transparently change variables and execution paths of Android malware, and to transparently repair realtime firmware. Together, these three components form a cohesive solution to the debugging transparency problem.

References

[1] CVE-2000-0573: Format string vulnerability. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2000-0573, 2000.

[2] CVE-2002-1496: Heap-based buffer overflow. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2002-1496, 2002.

[3] nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the 4th European Workshop on System Security (EUROSEC '11)* (2011).

[4] Adams, K. BluePill detection in two easy steps. http://x86vmm.blogspot.com/2007/07/bluepill-detection-in-two-easy-steps.html, 2007.

[5] Altera Corporation. PCI Express High Performance Reference Design. http://www.altera.com/literature/an/an456.pdf, 2014.

[6] Anubis. Analyzing unknown binaries. http://anubis.iseclab.org.

[7] Appel, A. W. *Compiling with continuations.* Cambridge University Press, 2007.

[8] ARM. DSTREAM. http://ds.arm.com/ds-5/debug/dstream/.

[9] Asset Intertech. Processor-Controlled Test. http://www.asset-intertech.com/Products/Processor-Controlled-Test.

[10] Aumaitre, D., and Devine, C. Subverting windows 7 x64 kernel with dma attacks. *HITBSecConf 2010 Amsterdam 29* (2010).

[11] Auty, M., Case, A., Cohen, M., Dolan-Gavitt, B., Ligh, M. H., Levy, J., and Walters, A. Volatility framework - volatile memory extraction utility framework.

[12] Azab, A. M., Ning, P., Wang, Z., Jiang, X., Zhang, X., and Skalsky, N. C. Hypersentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 38–49.

[13] Bachaalany, E. Detect if your program is running inside a Virtual Machine. http://www.codeproject.com/Articles/9823/Detect-if-your-program-is-running-inside-a-Virtual.

[14] Baecher, P., and Koetter, M. Libemu-x86 shellcode emulation library. *See http://libemu. carnivore. it* (2007).

[15] Baliga, A., Ganapathy, V., and Iftode, L. Automatic inference and enforcement of kernel data structure invariants. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual* (2008), IEEE, pp. 77–86.

[16] Bayer, U., Moser, A., Kruegel, C., and Kirda, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[17] Bianchi, A., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Blacksheep: Detecting compromised hosts in homogeneous crowds. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 341–352.

[18] Branco, R., Barbosa, G., and Neto, P. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. In *Black Hat* (2012).

[19] Bruening, D., Kiriansky, V., Garnett, T., and Amarasinghe, S. Dynamorio: An infrastructure for runtime code manipulation.

[20] Carrier, B. The Sleuth Kit. http://www.sleuthkit.org/sleuthkit/desc.php.

[21] Carrier, B. D., and Grand, J. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation 1*, 1 (2004), 50–60.

[22] checkvm: Scoopy doo. http://www.trapkit.de/research/vmm/scoopydoo/scoopy_doo.htm.

[23] Chen, X., Andersen, J., Mao, Z., Bailey, M., and Nazario, J. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the 38th Annual IEEE International Conference on Dependable Systems and Networks (DSN '08)* (2008).

[24] Damballa. Damballa State of Infections Report Q4 2014. `https://www.damballa.com/state-infections-report-q4-2014/`, 2014.

[25] Deng, Z., Zhang, X., and Xu, D. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'13)* (2013).

[26] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)* (2008).

[27] Dinaburg, A., Royal, P., Sharif, M., and Lee, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), ACM, pp. 51–62.

[28] Dolan-Gavitt, B., Leek, T., Hodosh, J., and Lee, W. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 839–850.

[29] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Security and Privacy (SP), 2011 IEEE Symposium on* (2011), IEEE, pp. 297–312.

[30] Dornseif, M. 0wned by an ipod. *Presentation, PacSec* (2004).

[31] Duflot, L., Perez, Y.-A., Valadon, G., and Levillain, O. Can you still trust your network card. *CanSecWest/core10* (2010), 24–26.

[32] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (2012), 6.

[33] Falliere, N. Windows anti-debug reference. `http://www.symantec.com/connect/articles/windows-anti-debug-reference`, 2010.

[34] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. Dynamic and Transparent Analysis of Commodity Production Systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10)* (2010).

[35] Ferrie, P. Attacks on more virtual machine emulators. *Symantec Technology Exchange* (2007).

[36] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 586–600.

[37] Gander, K. Drone delivering asparagus to dutch restaurant crashes and burns. `http://www.independent.co.uk/life-style/food-and-drink/news/drone-delivering-asparagus-to-dutch-restaurant-crashes-and-burns-10179731.html`, 2015.

[38] Garfinkel, T. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS* (2003), vol. 3, pp. 163–176.

[39] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. Compatibility is not transparency: Vmm detection myths and realities. In *HotOS* (2007).

[40] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 193–206.

[41] Garfinkel, T., Rosenblum, M., et al. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium* (2003).

[42] GNU. GDB: GNU Project Debugger. www.gnu.org/software/gdb.

[43] Guarnieri, C., Tanasi, A., Bremer, J., and Schloesser, M. The cuckoo sandbox, 2012.

[44] Hay, B., and Nance, K. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review 42*, 3 (2008), 74–82.

[45] Heasman, J. Implementing and detecting a pci rootkit. *Retrieved February 20*, 2007 (2006), 3.

[46] Holllander, R., and Bolotoff, P. RAMSpeed, a cache and memory benchmarking tool. http://alasir.com/software/ramspeed, 2011.

[47] IDA Pro. www.hex-rays.com/products/ida/.

[48] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[49] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on* (2014), IEEE, pp. 605–620.

[50] Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), ACM, pp. 128–138.

[51] Kaspersky Lab. Kaspersky Security Bulletin 2014. http://securelist.com/analysis/kaspersky-security-bulletin/68010/kaspersky-security-bulletin-2014-overall-statistics-for-2014/.

[52] Kaspersky Lab. Financial Cyberthreats. http://cdn.securelist.com/files/2015/02/KSN_Financial_Threats_Report_2014_eng.pdf, 2014.

[53] Kaspersky Lab. Kaspersky Lab Uncovers "The Mask": One of the Most Advanced Global Cyber-espionage Operations to Date Due to the Complexity of the Toolset Used by the Attackers. http://kaspersky.com/, 2014.

[54] Kirat, D., Vigna, G., and Kruegel, C. BareBox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)* (2011).

[55] Krishnan, S., Snow, K. Z., and Monrose, F. Trail of bytes: efficient support for forensic analysis. In *Proceedings of the 17th ACM conference on Computer and communications security* (2010), ACM, pp. 50–60.

[56] Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., and Ioannidis, S. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)* (2013).

[57] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on 38*, 1 (2012), 54–72.

[58] Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'11)* (2011).

[59] Lengyel, T. K., Neumann, J., Maresca, S., Payne, B. D., and Kiayias, A. Virtual machine introspection in a hybrid honeypot architecture. In *CSET* (2012).

[60] Lin, Z., Rhee, J., Wu, C., Zhang, X., and Xu, D. Dimsum: Discovering semantic data of interest from un-mappable memory with confidence. In *Proc. ISOC Network and Distributed System Security Symposium* (2012).

[61] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices* (2005), vol. 40, ACM, pp. 190–200.

[62] Mankin, J., and Kaeli, D. Dione: a flexible disk monitoring and analysis framework. In *Research in Attacks, Intrusions, and Defenses*. Springer, 2012, pp. 127–146.

[63] Martin, A. Firewire memory dump of a windows xp computer: a forensic approach. *Black Hat DC* (2007), 1–13.

[64] McAfee. Threats Report: Fourth Quarter 2014. `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2014.pdf`.

[65] Molina, J., and Arbaugh, W. Using independent auditors as intrusion detection systems. In *Information and Communications Security*. Springer, 2002, pp. 291–302.

[66] Moon, H., Lee, H., Lee, J., Kim, K., Paek, Y., and Kang, B. B. Vigilare: toward snoop-based kernel integrity monitor. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 28–37.

[67] Necula, G. C., McPeak, S., Rahul, S. P., and Weimer, W. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction* (2002), Springer, pp. 213–228.

[68] Payne, B. D. Libvmi: Simplified virtual machine introspection.

[69] Petroni Jr, N. L., Fraser, T., Molina, J., and Arbaugh, W. A. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium* (2004), San Diego, USA, pp. 179–194.

[70] Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., and Ioannidis, S. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security* (2014), ACM, p. 5.

[71] Quist, D., Smith, V., and Computing, O. Detecting the presence of virtual machines using the local data table. *Offensive Computing* (2006).

[72] Quist, D., and Val Smith, V. Detecting the Presence of Virtual Machines Using the Local Data Table. `http://www.offensivecomputing.net/`.

[73] Quynh, N., and Suzaki, K. Virt-ICE: Next-generation Debugger for Malware Analysis. In *In Black Hat USA* (2010).

[74] Raffetseder, T., Kruegel, C., and Kirda, E. Detecting system emulators. In *Information Security*. Springer Berlin Heidelberg, 2007.

[75] Reuben, J. S. A survey on virtual machine security. *Helsinki University of Technology 2* (2007), 36.

[76] Rutkowska, J. Red Pill. `http://www.ouah.org/Red_Pill.html`.

[77] Rutkowska, J. Beyond the cpu: Defeating hardware based ram acquisition. *Proceedings of BlackHat DC 2007* (2007).

[78] Sanabria, A. Malware Analysis: Environment Design and Architecture. `http://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841`, 2007.

[79] Snow, K. Z., Krishnan, S., Monrose, F., and Provos, N. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium* (2011).

[80] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS'08)* (2008).

[81] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*. Springer, 2008, pp. 1–25.

[82] Spensky, C. Analysis Time for Malware Samples. Email correspondence with author, 2015.

[83] Srinivasan, D., Wang, Z., Jiang, X., and Xu, D. Process out-grafting: an efficient out-of-vm approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security* (2011), ACM, pp. 363–374.

[84] Stewin, P., and Bystrov, I. Understanding dma malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2013, pp. 21–41.

[85] Tang, A., Sethumadhavan, S., and Stolfo, S. Unsupervised anomaly-based malware detection using hardware features. In *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds., vol. 8688 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 109–129.

[86] Wang, J., Stavrou, A., and Ghosh, A. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection* (2010), Springer, pp. 158–177.

[87] Wang, J., Zhang, F., Sun, K., and Stavrou, A. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on* (2011), IEEE, pp. 1–5.

[88] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., and Zeller, A. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis* (2010), ACM, pp. 61–72.

[89] Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering* (2009), IEEE Computer Society, pp. 364–374.

[90] Whelan, R., Leek, T., and Kaeli, D. Architecture-independent dynamic information flow tracking. In *Compiler Construction* (2013), Springer, pp. 144–163.

[91] Xiao, J., Xu, Z., Huang, H., and Wang, H. Security implications of memory deduplication in a virtualized environment. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (2013).

[92] Yan, L.-K., Jayachandra, M., Zhang, M., and Yin, H. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)* (2012).

[93] Yason, M. The art of unpacking. *Black Hat Briefings USA, Aug 2007* (2007).

[94] Yuschuk, O. OllyDbg. `www.ollydbg.de`.

[95] Zhang, F., Leach, K., Sun, K., and Stavrou, A. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (2013).

[96] Zhang, X., van Doorn, L., Jaeger, T., Perez, R., and Sailer, R. Secure coprocessor-based intrusion detection. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (2002), ACM, pp. 239–242.

A Appendix

Table 4: Summary of Anti-debugging, Anti-VM, and Anti-emulation Techniques

Anti-debugging [18, 33]	
API Call	Kernel32!IsDebuggerPresent returns 1 if target process is being debugged
	ntdll!NtQueryInformationProcess: ProcessInformation field set to -1 if the process is being debugged
	kernel32!CheckRemoteDebuggerPresent returns 1 in debugger process
	NtSetInformationThread with ThreadInformationClass set to 0x11 will detach some debuggers
	kernel32!DebugActiveProcess to prevent other debuggers from attaching to a process
PEB Field	PEB!IsDebugged is set by the system when a process is debugged
	PEB!NtGlobalFlags is set if the process was created by a debugger
Detection	ForceFlag field in heap header (+0x10) can be used to detect some debuggers
	UnhandledExceptionFilter calls a user-defined filter function, but terminates in a debugging process
	TEB of a debugged process contains a NULL pointer if no debugger is attached; valid pointer if some debuggers are attached
	Ctrl-C raises an exception in a debugged process, but the signal handler is called without debugging
	Inserting a Rogue INT3 opcode can masquerade as breakpoints
	Trap flag register manipulation to thwart tracers
	If entryPoint RVA set to 0, the magic MZ value in PE files is erased
	ZwClose system call with invalid parameters can raise an exception in an attached debugger
	Direct context modification to confuse a debugger
	0x2D interrupt causes debugged program to stop raising exceptions
	Some In-circuit Emulators (ICEs) can be detected by observing the behavior of the undocumented 0xF1 instruction
	Searching for 0xCC instructions in program memory to detect software breakpoints
	TLS-callback to perform checks
Anti-virtualization	
VMWare	Virtualized device identifiers contain well-known strings [23]
	checkvm software [22] can search for VMWare hooks in memory
	Well-known locations/strings associated with VMWare tools
Xen	Checking the VMX bit by executing CPUID with EAX as 1 [3]
	CPU errata: AH4 erratum [3]
Other	Apparent size of TLB changes in presence of VMM [4]
	LDTR register [72]
	IDTR register (Red Pill [76])
	Magic I/O port (0x5658, 'VX') [54]
	Invalid instruction behavior [13]
	Using memory deduplication to detect various hypervisors including VMware ESX server, Xen, and Linux KVM [91]
Anti-emulation	
Bochs	Visible debug port [23]
QEMU	`cpuid` returns less specific information [92]
	Accessing reserved MSR registers raises a General Protection (GP) exception in real hardware; QEMU does not [74]
	Running instructions longer than 15 bytes raises a GP exception in real hardware [74]
	Undocumented `icebp` instruction hangs in QEMU [92], while real hardware raises an exception
	Unaligned memory references raise exceptions in real hardware; unsupported by QEMU [74]
	Bit 3 of FPU Control World register is always 1 in real hardware, while QEMU contains a 0 [92]
Other	Using CPU bugs or errata to create CPU fingerprints via public chipset documentation [74]

Table 5: Description of test cases used in our experiments. The "# calls" row describes the total number of function calls made during the execution of that test case.

	# calls	Description
nullhttpd		
test 1	239	GET standard HTML page
test 2	239	GET empty file
test 3	239	GET invalid file
test 4	240	GET binary data (image)
test 5	245	GET directory
test 6	385	POST information
test 7	180	Exploit: buffer overrun [2]
wuftpd		
test 1	407	Change directory
test 2	453	Get /etc/passwd, text
test 3	457	Get /bin/dmesg, binary
test 4	22	Attempt executing binary
test 5	267	Login with invalid user
test 6	22	Exploit: format string [1]