Towards Transparent Introspection

Kevin Leach
University of Virginia
MIT Lincoln Laboratory
kjl2y@virginia.edu

Chad Spensky
University of California, Santa Barbara
MIT Lincoln Laboratory
cspensky@cs.ucsb.edu

Westley Weimer
University of Virginia
weimer@virginia.edu

Fengwei Zhang
Wayne State University
fengwei@wayne.edu

***Abstract*—There is a growing need for the dynamic analysis of sensitive systems that do not support traditional debugging or emulation environments. Analysis can alter program behavior, necessitating transparency. For example, as the cat and mouse game between malware authors and malware analysts progresses, malicious software can increasingly detect and confound debuggers. Analysts must understand variable values, stack traces, and factors influencing dynamic behavior, but recent malware samples leverage any piece of information or artifact available that signals the presence of a debugger or emulator. In this work, we advance the state-of-the-art for transparent program analysis by introducing a low-artifact introspection technique. Our approach uses hardware-assisted live memory snapshots of process execution on native targets (e.g., x86 processors), coupled with static reasoning about programs. We produce high-fidelity data and control flow information with minimal detectable artifacts that could influence benign subject behavior or be leveraged for anti-analysis. We evaluate our system using two hardware implementations (x86-supported System Management Mode and PCI-based SlotScreamer devices) and two software configurations (benign and evasive programs). We also analyze the theoretical and practical limitations of our technique. We discuss an expert case study in which we apply our technique to a malware reverse engineering task. Finally, we present results of a human study in which 30 participants performed debugging tasks using information provided by our approach; our tool was as useful as a gdb baseline, but applies transparently. Our dynamic analysis approach permits transparent introspection to access previously-unavailable information about a process's internal state with minimal instrumentation artifacts.**

I. Introduction

From embedded domains to virtualization to security, many software systems that require dynamic analysis cannot use traditional debuggers. Standard approaches, such as `gdb`, incur significant overhead or pause the subject program entirely, which can interfere with normal execution or timing. Many hardware-based approaches, such as the JTAG [39] standard for embedded testing, may introduce delays or be inapplicable in secure settings [39]. Advances in trace-based [38] or replay [43], [48] debugging often have lower overhead, but such post-mortem analyses can be too late for many issues. The essential problem here is that of the *heisenbug*, in which the very act of debugging incurs overhead or perturbs the system and thus changes the behavior to be studied [33]. We use the general term *program introspection* to encompass standard debugging and analysis actions, such as observing variable values, inspecting dynamic stack traces, or determining what influences the conditional behavior of unknown software.

We say that a dynamic analysis system, such as a debugger or introspection tool, is *transparent* if computational and environmental observations made by the subject system do not differ between analyzed runs and unanalyzed runs. A system that is not transparent instead introduces *artifacts* that can reveal that system's presence. Artifacts can be direct, such as the Windows API `isDebuggerPresent` method, or indirect, such as measurements of elapsed times between operations.

Two factors have led to a recent surge of interest in transparent process analysis techniques. First, hardware advances have made memory analysis techniques practical, low-overhead, and affordable [44], [53]. Second, the twin rise of virtualization and appearance of environment-aware malware [6], [9], [12], [13], [19] mean that there are many more systems that require transparent dynamic analyses. DARPA's recent announcement of the Transparent Computing Program [15] emphasizes the need for such techniques. An effective introspection system in this context must be transparent, provide accurate information, and support analysis tasks.

We propose a system that provides transparent introspection capabilities by asynchronously acquiring snapshots of program memory at runtime. Given such snapshots, we bridge the *semantic gap* (i.e., reconstruct important program data), converting from operating system information to process address spaces to variables, buffers, and stack frames of interest to the analyst. With the advent of new commodity hardware and virtual machine techniques capable of quickly reading system memory, we believe a transparent program introspection framework is now possible. We propose and evaluate HOPS, a lightweight, native, accurate, asynchronous introspection technique that usefully supports analysis tasks.

We evaluate HOPS with respect to those three properties. We consider the transparency of our system and the artifacts it might produce, with a focus on time overhead. We measure the accuracy of our system's variable value and stack trace information against ground truth. Finally, we conduct a human study of 30 participants to evaluate our system's ability to support standard debugging tasks, as well as an expert case study to evaluate our system's ability to support a malware reverse engineering task. We find that HOPS is generally accurate for variable values and stack traces, is able to support conventional maintenance tasks as well as `gdb`, and supports domain-specific reverse engineering tasks. In addition, because we focus on zero-overhead approaches, HOPS can be used in systems where traditional debuggers cannot apply.

II. Background

In this section, we introduce the hardware support that permits an effective implementation of HOPS.

A. System Management Mode

System Management Mode (SMM) [25] 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. SMM is triggered by asserting the System Management Interrupt (SMI) pin on the CPU. This pin can be triggered in a variety of ways, which include writing to an I/O 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[1]. Thus, SMRAM can 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 any instructions requiring any privilege level. The `RSM` instruction forces the CPU to exit from SMM and resume execution in the previous mode.

B. SlotScreamer

SlotScreamer [20] is a recently-introduced memory forensic device with software running on a USB3380 chip. The chip consists of a PCI Express connector that attaches to the motherboard of the system under test and a USB3 port connecting to a remote system. The device firmware can generate arbitrary PCI packets. The remote system fakes DMA packets to rapidly read memory from the system under test. This produces a *smear* of memory in which contents may change while being gathered. Using this technique is not perfectly transparent as it updates the DMA access performance counter. While SlotScreamer is a promising new device, it has not yet been formally evaluated in debugging scenarios.

III. Approach

A. Hops — *Transparent Process Introspection*

We present Hops, an approach for hardware-supported low-overhead asynchronous debugging. Transparency is critical for benign systems with heisenbugs as well as security-critical software or malware samples that may behave differently when executing in a debugger or virtual machine framework. Essentially, our approach iteratively inspects snapshots of physical memory (recorded via special hardware) and then uses a combination of operating system and compiler techniques to locate local, global, and stack-allocated variables and their values as well as to determine the current call stack. Because we may operate on off-the-shelf optimized code and only assume access to memory (e.g., we cannot access CPU registers), our approach may not be able to report the values of some variables (such as those that are stored in registers) or some stack frames (such as those associated with functions that have been inlined). Nonetheless, we demonstrate empirically in Section V that our approach provides rich information for common maintenance and security analysis tasks in practice.

1) Input Assumptions: We detail the assumptions of our approach that delineate its applicability.

Assembly Code. We assume access to the assembly code of the target executable, but not knowledge of the exact assembler or compiler flags or options used to produce that executable. For example, we may know that the target machine is running a particular version of the Apache HTTP Server, but not whether the deployed version was compiled with "–O2" or "–O3". We can make use of the source code, when applicable.

Memory Snapshots. Most critically, we assume access to periodic samples or snapshots of physical memory. Such snapshots could be provided via a hardware-based system (e.g., [7], [11], [14], [46], [50]), or through virtual machine introspection. In Section V, we evaluate our approach using both x86-supported System Management Mode and also PCI-based SlotScreamer devices to gather memory snapshots.

We focus on PCI-e as the ideal candidate for memory introspection because of its promising throughput rates. The rate at which we can obtain snapshots is limited theoretically by peak PCI-e transfer speeds (i.e., to 15.754GB/s, or roughly to 3.85 million page samples per second). In practice, however, most implementations can expect to achieve an effective transfer rate of 250MB/s per PCI lane, or 4GB/s total using 16 lanes [3]. This equates to 1.05 million page samples per second, or roughly 3200 cycles per page acquisition on our experimental platform (see Section V). We evaluate using SlotScreamer, which is a special PCI-express board with a USB3 interface permitting DMA access to a host's memory from a remote system [20]. SlotScreamer permits acquiring smears of memory very rapidly (theoretically, it is band-limited by USB3, which permits reading memory at 4.8Gbps or roughly 146 page samples per second).

Alternatively, we can acquire snapshots via System Management Mode on x86 machines. SMM remains available on extant and current x86 systems, admitting broad applicability. Using SMM allows the capture of pages of memory at a time, albeit with high overhead (roughly 12μs to capture a page of memory).

In either case, we seek to reconstruct useful semantic information from these acquired snapshots.

Normal Systems. We do not consider any modification to the target machine or the target executable beyond the hardware required to log memory snapshots. That is, one plug-in PCI device for SlotScreamer or an unchanged x86 processor for SMM. In particular, we do not replace, patch, or otherwise change the target executable or the OS on the target machine.

2) Architecture: Our approach follows a pipelined architecture in which raw snapshots of physical memory are passed through a number of analyses, each of which refines or approximates the information available. The raw snapshots are collected at regular intervals. We focus on reporting the values of variables (local, global, or stack-allocated) as well as determining the dynamic stack trace. Figure 1 illustrates the proposed approach. We start with raw snapshots and combine existing OS introspection techniques (e.g., Spectre [52], Volatility [5]) to reason about variable and stack trace information in a particular program being tested (denoted Code Under Test in the figure).

3) Physical Memory Snapshots: First, we direct the memory snapshot hardware to log physical pages related to the

[1] Requests for SMRAM addresses forward to video memory by default.

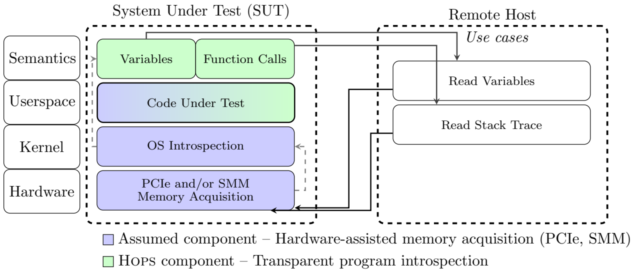


Fig. 1. Architecture of the proposed HOPS system. We assume access to hardware that can transparently acquire snapshots of system memory from the System Under Test. Combined with OS introspection techniques [5], [52] to find the program being tested (Code Under Test), we use these snapshots to reason about variable and stack trace information in the Code Under Test on an external system (Remote Host).

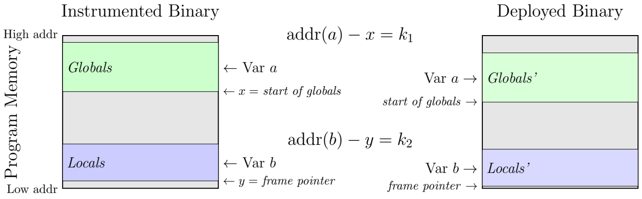


Fig. 2. We hypothesize for some binaries that two variables exist at the same offsets between two different compiled version. In both versions, we hypothesize that a variable a exists at the same offset from the start of the globals (x in the figure). Similarly, we also hypothesize that stack allocated variables (b) exist at a fixed offset from the frame pointer (y).

target executable. This is a matter of bridging the semantic gap between raw bit patterns and logical program data and code. This is done by inspecting in-kernel data structures in physical memory to find the virtual address space mapping for the target process. From the virtual address space mapping, we can obtain a sequence of physical pages that correspond to the process's virtual address space. In addition, from in-kernel data structures, such as the process control block, we note the memory ranges associated with the stack segment, data segment, and code segment. Interpretation of relevant kernel data structures is well-described in the semantic gap literature [22], [26], [29], [32], [52]. HOPS can use any such black-box analysis to bridge the semantic gap.

4) Reporting Variables: Given a correctly-ordered snapshot of a process's virtual address space as well as the locations of the various segments, our analysis proceeds by enumerating hypotheses about the locations of variables. These hypotheses are heuristically ordered by decreasing level of confidence. We start with the standard techniques used by debuggers.

For global variables, we use information from the symbol table (e.g., PE in memory for Windows executables, ELF for Linux). For locals, we focus on variables that were stack allocated (by the writer of the assembly code or by the compiler). We use debugging information when available.

Some variables do not admit localization in this manner (e.g., variables that are stored in registers). In such cases, our second hypothesis is based on the relative location of the those variables in an instrumented binary on a test machine. Recall that we do not assume knowledge of the exact assembler, linker or compiler flags used to produce the target executable. Thus, we track the relative locations of variables (e.g., global a is stored at offset x from the data segment while local variable b is stored at offset y from the frame pointer) and hypothesize that those offsets will be the same for the target executable. This hypothesis does not always hold, but it allows us to recover information for additional variables in practice. Figure 2 explains this hypothesis visually.

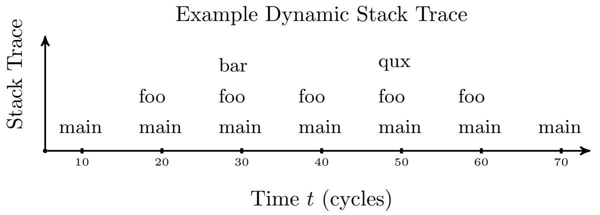


Fig. 3. Example ground truth dynamic stack trace. The dynamic stack trace is a time series of static call stacks showing which functions are called over time in the program.

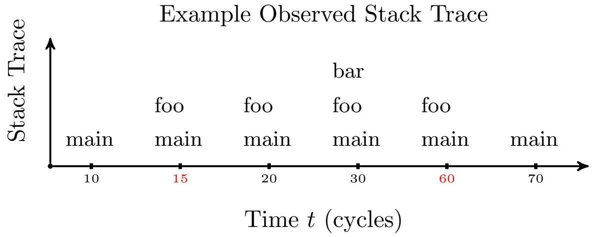


Fig. 4. This example set of stack trace observations demonstrates the tradeoffs between accuracy and transparency. Sampling too frequently may require additional resources or introduce artifacts without yielding additional information (e.g., sampling at $t = 15$ and $t = 20$). Conversely, sampling too coarsely may miss function calls (e.g., between $t = 30$ and $t = 60$).

5) Reporting Stack Traces: In addition to the values of particular local and global variables, we also produce snapshots of the stack trace and the values of variables in caller contexts. We consider a number of standard calling conventions (e.g., stdcall, cdecl, fastcall) and attempt to locate the chain of activation records on the stack following standard debugger techniques [51]. For example, the name of the calling function is determined by finding the return address on the stack and mapping it to the nearest enclosing entry point in the code segment via the symbol table.

We also consider samples for which there is no source available. We can operate on labeled disassembly, as produced by IDA Pro or a similar tool. In such cases, synthetic label names or unique heuristic names are used [24] (e.g., "`printf`-like function #2" may label a function that behaves like the `printf` function). While walking the stack, we gather hypotheses about the locations and values of stack-allocated variables (such as the actual arguments or locals in-scope in parent contexts), as above.

Figures 3 and 4 illustrate our notion of a dynamic stack trace and the potential tradeoffs between transparency and introspection accuracy. We refer to the sequence of activation records (i.e., stack frames) present at a particular point in a program's execution as a static call stack. A *dynamic stack trace* of a program is a time series of static call stacks. Ideally, a dynamic stack trace includes a new static stack frame every time the subject program calls or returns from a procedure. Recording the static call stack after every instruction would observe every call and return, but would likely be resource- and timing-intensive and introduce anti-analysis artifacts. Conversely, recording too few static call stacks results in a dynamic stack trace that may miss important behavior.

6) Output: One basic output of our technique is a dynamic stack trace: A sequence of static call stacks, one call stack per snapshot of physical memory. Each call stack lists the name of each called function and the values of its actual arguments. In addition, the analyst can request to inspect the value of a particular local, global, or stack-allocated variable. Both call stack reports and variable value inspections are asynchronous.

Note that any presented information may be incorrect. For example, no information may be available about variables optimized away by the compiler or stored in registers. However, we hypothesize that optimized-away variables are not implicated in standard maintenance or security use-cases. For example, if the variable `x` is optimized away after `x=4;`, a developer with access to the source code may be able to reason about conditions involving `x` even if HOPS cannot report its value. In security settings, if a variable in the source is optimized away and not present in the deployed code, it is unlikely that that variable could be used maliciously.

Similarly, a hand-crafted leaf function that uses a non-standard calling convention (e.g., a custom hybrid of callee-saves and caller-saves for registers without pushing the return address) may not show up on a heuristic stack trace. However, such behavior is not commonly observed in general software systems (e.g., it is not easy to express in standard C), and in particular is not common in malware in the wild: stack and heap overflow, heap spray, and return-oriented programming (ROP [34]) and jump-oriented programming (JOP [8]) attacks depend on the traditional stack model for both Linux and Windows platforms.

IV. USE CASES AND PROTOCOLS

In this section, we describe intended use cases for HOPS that set the stage for restrictions about the environment used by our algorithm. We describe the general implementation of the algorithm and protocol used to conduct our human study. Experiment-specific details are described in Section V.

We envision three use cases for our system. In all 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 first and second cases but unlikely in the third), we take advantage of it to formulate additional hypotheses about variable locations (see Section III-A4). In all cases, 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.

UC1 — Maintenance Analysis of Benign Binaries. In our standard use case, HOPS supports the maintenance of non-malicious software by providing information about variable values and stack traces. We consider standard maintenance tasks such as fault localization, refactoring, or debugging that would normally be supported by a tool such as `gdb`. In this use case, the source code is likely to be available, but heisenbugs, timing dependencies, or similar issues still require the use of a transparent analysis technique. The primary metric is the fraction of maintenance questions the analyst is able to answer correctly when supported by information from HOPS. Secondary metrics include HOPS's accuracy when reporting variable values and stack traces, and HOPS's transparency.

UC2 — Security Analysis of Benign Binaries. The second use case is for software deployed in large enterprises where multiple instances of the same software are running on multiple hosts and an exploit occurs. In this case, we have vulnerable software running and, as the exploit occurs, we want to know which memory locations are implicated. 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 speed of our asynchronous debugging approach: for example, we may require that information about potentially-malicious data be available quickly enough to admit classification by an anomaly intrusion detection system.

UC3 — Security Analysis of Malicious Binaries. In automated malware triage systems, we desire to analyze a large corpus of malware samples as quickly as possible. Unfortunately, existing solutions to this problem depend on virtualization. For example, the common Anubis [4] framework, which analyzes binaries for malicious behavior, depends on Xen for virtualization, which allows stealthy or VM-aware malware to subvert the triage system. In contrast, our system assumes the presence of low-overhead sampling hardware that enables fast access to a host's memory. Our algorithm is thus charged with introspecting the malware process when running in the triage system. In such a triage system, we want to understand a sample's behavior in part by knowing the values of variables and producing a dynamic stack trace as the sample executes. For example, the system might inspect control variables and function calls to determine how the malware sample detects virtualization or might inspect snapshots of critical but short-lived buffers for in-memory keys. In this use case, the primary metric of concern is the fraction of critical malware aspects (e.g., artifacts used to evade analysis) an analyst can identify while supported by information from HOPS. A secondary metric is accuracy with respect to variable values and stack trace information.

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

Human Study Protocol

The goal of our human study is to measure how well humans can perform debugging and maintenance tasks when supported by HOPS—exploring our first use case, the maintenance analysis of benign programs. Simply put, we presented each participant with a snippet of code and the output from either HOPS or `gdb`. We then measured participant accuracy on maintenance questions regarding that snippet. Multiple snippets were shown to each participant in a survey.

Participants were presented instructions for completing the survey, as well as example questions and possible answers. This training helps address mistakes attributed to confusion or training effects. Each snippet was shown with corresponding output from a debugging tool—either from HOPS or `gdb`—randomly selected for each participant on each question. Additionally, each snippet was shown with a corresponding question meant to test understanding of the snippet during execution. Participants were asked to answer the question in free form text. Finally, participants were presented with an exit survey asking for personal opinions on the debugging tools and experience. We describe participant selection, snippet selection, and question selection in detail.

7) Participant Selection: We require participants that have at least novice software development skills. We solicited responses from 24 third and fourth year undergraduate students enrolled in a computer security course and 6 graduate students. Participants were kept anonymous and were offered a chance to win one of two $50 Amazon gift cards or class extra credit (via randomized completion codes). We removed participants from consideration if they scored more than one standard deviation below the average score or if they failed to provide responses to all questions. We impose this restriction due to the difficulty of controlling for C development and debugging experience. Participants were made aware of these requirements and that their potential reward depended on it.

8) Snippet Selection: The goal of the human study was to simulate debugging or maintenance in a controlled environment. We selected snippets of code from the two open source projects, *nullhttpd* 0.5.0 (1861 LOC) and *wu-ftpd* 2.6.0 (29,167 LOC). To create a snippet, we first randomly selected a function defined in the source code of each project. Only functions that were at least 10 lines long and were reachable by one of our test cases were considered. Similarly, functions longer than 100 lines were truncated to their first 100 lines. We then chose a random reachable point within that function.

The snippet thus consisted of that function, with the particular reachable point visibly marked as a breakpoint—as if the participant had placed a breakpoint on that line in a debugger and run the program until the breakpoint was reached and execution paused. Every snippet corresponded to a point by the test suite, debugging information was obtained by running the program on the test suite and invoking `gdb` or HOPS at that point. Ultimately, 23 snippets were created. Snippet counts and size limits were selected to ensure a reasonable completion time by the participants.

9) Software Maintenance Questions: This study measures how the information provided by our technique aids a developer when reasoning about code. We require participants to answer questions that are indicative of debugging activities that developers might ask during the maintenance process. Sillito et al. identify several different types of questions real developers ask during maintenance tasks [41]. Following previous human study protocols involving software maintenance [21], we used these three human study questions (HSQ):

HSQ1 What conditions must hold true to reach line X during execution?
HSQ2 What is the value of variable "`y`" on line X?
HSQ3 At line X, which variables are in scope?

Many questions discussed by Sillito et al. were general in nature and would not have been applicable for gauging participants' understanding of the snippets used in the study (e.g., one question reads "Does this type have any siblings in the type hierarchy?", which is not applicable to our subject C programs). Questions are randomly assigned to each snippet.

V. EVALUATION

We consider four primary research questions when evaluating HOPS.

RQ1. On average, what fraction of local, global, and stack-allocated variable values can our system correctly report under multiple hardware regimes?

RQ2. How accurately can our system correctly report dynamic stack traces as a function of the asynchronous sampling rate of memory snapshots?

RQ3. Is the information provided useful for reasoning about debugging tasks compared to the state of the art?

RQ4. Could the information provided by our system help analysts reason about VM-aware malware?

At any given point in time, some subset of the target program's variables are available. For RQ1, we measure success at each time step in terms of the fraction of those variables for which our technique reports the correct value (w.r.t. ground truth). Similarly, at any given point in time during execution, there is a particular stack of activation records. We further evaluate the performance of our technique when implemented atop both SMM and SlotScreamer to establish our approach's feasibility on current hardware. For RQ2, we introduce a metric that requires functions to be reported and correctly ordered (w.r.t. ground truth). We then evaluate HOPS in terms of this metric as a function of the sampling rate (i.e., how often asynchronous memory snapshots are made available). For RQ3, we conduct a human study in which 30 participants answer debugging questions about snippets of code using information from HOPS and `gdb`. Finally, for RQ4, we consider a case study involving a VM-aware program sample that checks a number of different artifacts to detect analysis and report the fraction of those artifact queries that can be identified using HOPS.

A. Experimental Setup and Benchmarks

We evaluate HOPS 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. For nullhttpd, we consider a remote heap-based buffer overflow [2], while for wu-ftpd, we consider a format string vulnerability [1]. In addition to these exploits, for each program we consider non-malicious indicative test cases taken from previous research [28]. For example, one of the web server test cases retrieves a webpage, while one of the FTP server test cases logs in anonymously and transfers a file. Table I summarizes the test cases used in our experiments. As in Section IV, when the source code is available, our approach uses it to construct additional hypotheses about variable locations, but we do not assume that the compiler flags used in the deployed executable are known or the same. In these experiments, we simulate that disparity by gathering

TABLE I. Description of test cases used in our experiments.

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

hypotheses from programs compiled with "–O2" but evaluating against different binaries produced with "–O3".

Evaluating RQ1 and RQ2 requires that we establish ground truth values of variables and stack traces at every program point. For the purposes of evaluation only, we employ source-level instrumentation [31] to gather these values. Because our approach is based on memory snapshots and local, global, and stack-allocated variables, we instrument and evaluate at all points where a variable enters or leaves scope or is placed on the stack, including all function calls, function entry points, and loop heads. We also separately instrument for timing information, using the Intel `rdtsc` instruction. Recording the timing information for instrumentation points introduces a small overhead (2% on average on these benchmarks). Note that any instrumentation overhead applies only to gathering ground truth information for our experimental evaluation and is not part of our proposed algorithm.

We also note that, as with a standard debugger [24], [51], heap-allocated variables are accessed in our system by traversing expressions that start with local or global variables ("roots"). For example, after `glob_ptr = malloc(...)`, if an analyst wishes to inspect `glob_ptr->x`, the request is handled in four steps: 1) locate the (constant) address of `glob_ptr` in the data segment; 2) read the (dynamic) value stored there from the memory snapshot; 3) add the (constant) offset associated with `x`; and 4), read the (dynamic) value stored there from the memory snapshot. Accuracy on heap-allocated variable expressions thus reduces to accuracy on local, global, or stack-allocated variables.

In addition, for RQ4, we also consider *pafish* v04, 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. Evaluating RQ4 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.

Software and SlotScreamer experiments concerned with RQ1 and RQ2 were conducted on a 3.2GHz Intel Xeon X5672 machine with 48GB RAM. This system uses 64-bit Linux 3.2. SMM-based experiments were conducted using 32-bit Linux 2.6 on a system with an AMD Sempron CPU and 4GB RAM. In each case, we consider two versions of the same binary—ultimately, this means the runtime locations of variables will change due to address randomization. To facilitate experimental reproducibility and determinism, we disabled Address Space Layout Randomization (ASLR). However, it is possible to account for ASLR by bridging the semantic gap in kernel memory to find the offset used for randomization.

For RQ4, we ran pafish on a 64-bit Windows 7 system with two Intel Xeon E5-2660 CPUs at 2.2GHz and 48GB RAM.

TABLE II. Variable introspection accuracy.

	nullhttpd			wuftpd		
	Soft	SMM	PCI	Soft	SMM	PCI
Locals	43%	66%	41%	46%	48%	45%
Stack	65%	13%	58%	56%	31%	54%
Globals	100%	83%	96%	92%	93%	90%
Overall	69%	54%	65%	65%	57%	76%

B. RQ1 — Variable Value Introspection

We evaluate the accuracy of Hops with respect to asynchronous requests for variable values: what fraction of variables will our technique accurately report, averaged over every variable in scope at every function call, function entry, and loop head? To admit a more fine-grained analysis of our technique, we partition the set of all in-scope program variables into local, global, and stack-allocated. For this experiment, the set of *local* variables for a function is all locally-declared variables including those in various nested local scopes, as allowed in C. The set of *global* variables for a function is all global variables in scope at that function (e.g., not including global variables declared after that function or `static` variables in other modules). The set of *stack-allocated* variables of a function includes the function's arguments as well as the local and stack-allocated variables of all (transitive) callers of the function. We do not consider variables that are optimized away by the compiler because variables that are not present at run-time are unlikely to be implicated by exploits and may admit maintenance reasoning via context clues (see Section III for a qualitative explanation, as well as Section V-D for a quantitative justification). At each point, we query the value of each variable using our system and report the result. The result is correct if it matches the ground truth: for strings, this takes the form of a string comparison while all other variables (e.g., integers, pointers) are compared numerically.

Table II reports the results. The "Soft" columns record our accuracy via software simulation (rather than any special hardware). The "SMM" columns report our accuracy using x86 System Management Mode. The "PCI" columns report our accuracy using PCI-e based SlotScreamer hardware.

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). Specifically, 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 83–100% 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). Over the three snapshot-gathering techniques, HOPS answered 54–76% of variable introspection queries correctly. We consider what these accuracy results mean in the context of supporting software maintenance questions in Section V-D.

1) SMM Implementation: We used SMM to transparently collect pages of memory during execution of each program. We determine the location in physical memory of process pages using the CR3 register of the program under test as described in SPECTRE [52]. The SMI handler has access to the CR3 register of the currently executing process in SMRAM. From this value, we 1) find the circularly-linked list of processes (i.e., `task_struct`) in kernel memory, 2) iterate through the list to find the process under test, and 3) find the virtual memory mappings to help derive addresses of associated pages. We noted that the kernel could potentially swap processes out, in which case the mappings in the process's page table may not be directly applicable. For the purposes of experimental evaluation, we disable swapping as it is unlikely to be an issue in our proposed use cases. We then use our technique to determine the hypothesized location of each variable within the page. We compare this value against ground truth data and report the ratio of correctly identified variables. The results are similar overall to the software simulated and PCI-based results. The SMM platform shows worse performance with stack parameters passed to functions. This is most likely due to the differing calling conventions, compiler versions, and number of available registers between the two platforms used in these experiments.

2) SlotScreamer Implementation: The accuracy results when HOPS is deployed atop SlotScreamer are shown in the "PCI" columns in Table II. SlotScreamer transparently collects smears of memory during execution of each test program. We bridge the semantic gap as above. We use our technique to determine the hypothesized location of each variable in the test program within physical memory. We then use Inception [30] so that SlotScreamer acquires the memory smear. We compare each reported value against ground truth data and report the ratio of correctly identified variables. The results are comparable to the software simulation. SlotScreamer's higher performance on stack variables and lower performance on global variables is best explained by asynchronous updates to memory (memory smears are not atomic views of pages).

C. RQ2 — Stack Trace Introspection and Sampling Rate

For this research question we evaluate the portion of dynamic stack trace information that our technique can accurately report given a memory snapshot every k cycles. We report a single activation record as a tuple consisting of a function name and a list of actual argument values. A single static stack trace (at a given point in time) is thus a sequence (stack) of activation records.

We are ultimately interested in changes to the stack over time: a full dynamic stack trace is a sequence of static stack traces (each one corresponding to a point in time). For simplicity of presentation, we elide variable values (for which our accuracy is evaluated in RQ1) and denote a dynamic stack trace as a sequence of tuples (t, s) where t is the time in cycles and s the static call stack (e.g., $f_1 \rightarrow f_2 \rightarrow f_3$) corresponding to the activation records live at time t.

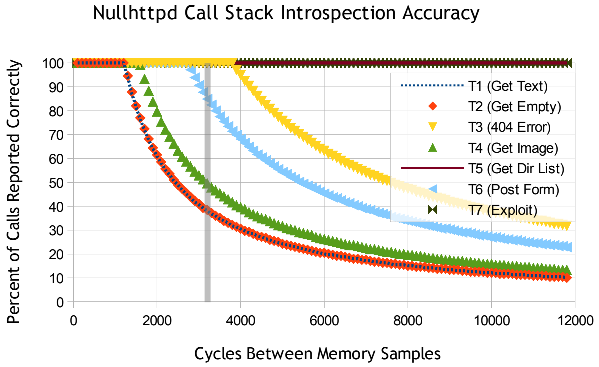


Fig. 5. Call stack introspection accuracy for Nullhttpd as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 1200 cycles yields perfect accuracy.

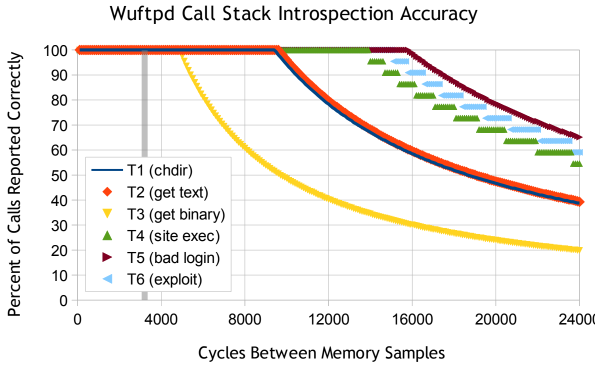


Fig. 6. Call stack introspection accuracy for Wuftpd as a function of the number of machine cycles between memory samples. The reference line at 3200 corresponds to current hardware. On all tests sampling every 4800 cycles yields perfect accuracy.

Our ground truth answer is equivalent to having a full memory snapshot available at every cycle. That is, a ground truth dynamic stack trace corresponds to 100% of the function calls that were invoked or returned from during that execution. For this experiment, we consider function calls only in userspace (e.g., program functions like `main` and library functions like `printf` are included, but actions taken by the kernel on behalf of system calls like `write` are not).

The metric used for this experiment is 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 omissions. Because the stack trace length differs among test cases and programs, we normalize this value to 100%. Thus, a stack trace identical to the ground truth corresponds to an accuracy of 100% while the example above missing 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., edit distance, LCS), we prefer this metric because it is conservative and corresponds to our algorithmic framework (see Figure 4).

Figure 5 reports the results for stack trace introspection for nullhttpd. As discussed in Section III, 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 time, 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 6 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, HOPS 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.

D. RQ3 — Human Study

We conducted a human study to measure how helpful or informative HOPS is to humans in practice. The study involved 30 participants (24 undergraduate and 6 graduate students). Each participant was shown 23 snippets of code and corresponding debugging output from either HOPS or `gdb`, and then asked a debugging question. Each question was randomly selected from a pool of three questions (HSQ1–3 in Section IV-9). We measured the accuracy of each participant on each question as well as the time taken to answer each question. We calculate accuracy by manually grading each participant's responses: each answer given by each participant is assigned a score from 0.0 to 1.0. For HSQ1 and HSQ3, the correct answer may consist of multiple parts (e.g., multiple conditions may be required to reach a particular line of code). For these cases, the participant's score is the fraction of correctly identified conditions or variables. For HSQ2, the participant's score is either 0 or 1.

We divide our human study results into two groups: the `gdb` group (control) and the HOPS group (treatment). We find that the control and treatment groups answered questions with an average accuracy of 59.2% and 68.0%, respectively[2]. The HOPS treatment group is at least as accurate as the control group with statistical significance ($p < .01$ using the Wilcoxon rank-sum test). Additionally, the control group took an average of 105±6.4 seconds to answer each question while the treatment group took an average of 118±8.2 seconds on each question. Differences in timing were not statistically significant.

Because humans are at least as accurate using HOPS when answering indicative software maintenance questions, we conclude that HOPS could usefully support standard software maintenance analysis tasks. We do not claim that HOPS is generally better than `gdb`—indeed, `gdb` has many features, such as changing variable values or poking memory, that HOPS does not support. However, HOPS is transparent, allowing it to be used in heisenbug or security-analysis tasks where `gdb` is inapplicable. Previously, developers had little to no information in such situations; HOPS transparently provides information that is accurate enough to usefully support analysis tasks.

TABLE III. LIST OF ARTIFACTS USED BY PAFISH.

Method or Artifact	HOPS Success
Debuggers	
IsDebuggerPresent	✗
CheckRemoteDebuggerPresent	✗
OutputDebugString	✓
General Sandboxes	
GetCursorPos	✓
GetUserName	✗
GetModuleFileName	✓
Disk legitimacy	✓
Disk size	✓
GetTickCount	✓
QEMU Registry Keys	
Device names	✓
BiosVersion	✓
Sandboxie	
sbiedll.dll	✗
VirtualBox	
Registry information	✓
Drivers	✓
MAC Address	✓
Window	✗
Processes	✓
VMWare	
Device names	✓
VMWare Tools	✓
Drivers	✓
Wine	
kernel32.dll features	✓
Hooking	
various	n/a

E. RQ4 — Pafish Case Study

This case study evaluates the utility of the information provided by our approach in assessing the artifacts inspected by VM-aware malware. We manually annotated pafish to collect ground truth data, as in RQ1 and RQ2.

Pafish is an open-source program for Windows that essentially runs through a list of checks to determine if various debuggers, emulators, or virtualization techniques are being employed by the system. For each check, the program reports whether or not the associated artifact was found.

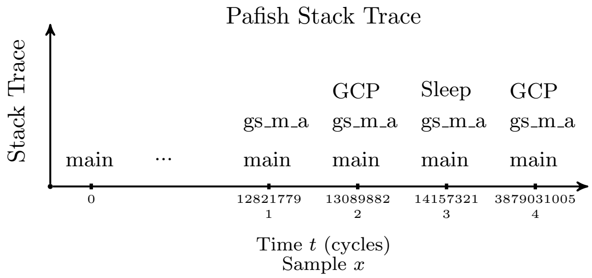
Pafish is particularly useful in evaluating HOPS because 1) it is is amenable to complete ground-truth annotation (unlike a "wild" malware sample, for which we could entirely miss a stealthy check and thus have false negatives) and 2) it helps answer RQ4 in a general manner (because it contains a large number of indicative artifact checks), which ultimately gives confidence that our tool applies to our considered use cases.

[2] Human study materials and anonymized responses are available at *http://church.cs.virginia.edu/hops-materials/all.tar.gz.*

Using HOPS, we can introspect visible variables and dynamic stack traces as in RQ1 and RQ2. We consider the question: "are the variables and stack traces values that HOPS reports accurate enough to conclude which anti-debugging techniques pafish is employing?" While analyst skill plays a role in such tasks, for this evaluation we used a conservative criterion, indicating success only for cases in which variables and function calls directly implicating the artifact were introspected correctly. For example, calling the `OutputDebugString` method in Windows would cause an error if a debugger is *not* attached. HOPS reports the call to `OutputDebugString` (ultimately culminating in a `write`-like system call), as well as its parameter (a *stack* variable in RQ1). From this, an analyst could accurately determine the artifact being employed in this scenario (i.e., `OutputDebugString`'s conditional behavior).

Table III summarizes which of the 22 artifacts considered by pafish can be detected with our stack tracing and variable introspection technique. A ✓ indicates that an analyst could use introspection information from HOPS to determine that pafish is using the given anti-analysis method or artifact. A ✗ indicates that perfect variable or stack trace introspection information would allow the analyst to determine that the given anti-analysis method is being used, but in practice HOPS does not provide accurate information about the relevant variables or stack frames (i.e., we cannot sample quickly enough with current hardware). For example, the `IsDebuggerPresent` API call is very fast. As a result, our current sampling rate is too coarse to capture the calls to this function. In fact, all five of the failing cases result from too coarse a sampling rate. In these situations, HOPS could acquire accurate stack traces, and thus implicate the artifacts, with a faster sampling rate (i.e., improved hardware). Finally, some methods or artifacts are beyond the scope of our technique. For instance, checking for hooked functions does not require calling any functions at all (instead, it scans virtual addresses of API functions for particular signatures, using values in registers). However, HOPS requires activation records created by function calls or variables stored in memory, so even with perfect memory introspection accuracy, HOPS could not reveal such artifact usage. We refer to these types of artifacts as being not applicable to our approach and denote them with an "n/a" in Table III.

Overall, Table III shows that we can accurately discern when pafish attempts to use 16 of the 22 artifacts in its suite. That is, for these 16 artifacts, our technique can be applied to provide useful information to an analyst who can conclude that a specific artifact was used. As an example, Pafish uses the `GetCursorPos` API call to determine the position of the mouse cursor. It calls this twice and concludes it is being analyzed if the mouse does not move at all after 2 seconds. Figure 7 shows a run of our system against pafish. We elide other function calls and focus specifically on the code near its use of `GetCursorPos`. We show that HOPS allows an analyst to see the calls to `GetCursorPos` and `Sleep` (samples 2, 3, and 4 in the Figure). At these points, we can also correctly introspect the values of the stack allocated variables `p1` and `p2`, which are pointers to `POINT` structures with x and y fields that correspond to the cursor's position. In summary, our system permits the analyst to see 1) the stack trace including the two GetCursorPos invocations, and 2) the variables in which the cursor's x and y positions are stored. This information implicates the exact artifact used (i.e., lack of mouse movement over time).



Code around sample 1

```
t = ...       int gensandbox_mouse_act(){
12821779        POINT p1, p2;
13089882        GetCursorPos(&p1);
14157321        Sleep(2000);
3879031005      GetCursorPos(&p2);
              if (p1.x==p2.x && ...)
                traced("found");
              else
3879559528      nottraced("not found");
```

Fig. 7. Stack trace gathered against pafish, specifically focused on the `GetCursorPos` artifact. The top of the figure shows the stack trace acquired by HOPS over time as a function of CPU clock cycles. At the bottom, the source code is annotated by the timestamp at which the line runs. At these points of time, we are capable of acquiring the values of structures `p1` and `p2`. The `traced` line is never executed during this run because the mouse was moved.

F. Evaluation Conclusions

This evaluation of our system against three programs (i.e., nullhttpd, wuftpd, and pafish) provides promising results in terms of accuracy and is indicative for the types of programs and workloads in our use cases. Our empirical results measure the tradeoff between introspection accuracy and transparency (sampling rate) for our low-artifact analysis technique. Essentially, HOPS constructs meaningful debugging information (variables and stack traces) from raw memory dumps provided via low artifact hardware (e.g., via PCI-e DMA). Recall that the ultimate goal of this system is to assist program analysis, whether for software maintenance, manual analysis or automated triage. In this regard, HOPS is 84% accurate over all variables and test cases considered (cf. Table II, but note that there are more globals and stack-allocated variables than local variables). Third, in a human study involving 30 participants, information provided by HOPS was no worse than information provided by `gdb` when supporting debugging questions with statistical significance. Lastly, we again demonstrate that HOPS is useful in practice by testing it against pafish, an open-source artifact detection platform. HOPS is capable of detecting when Pafish uses 16 out of 22 artifacts during its execution.

By implementing HOPS on two different hardware platforms, we demonstrated its generalizability under multiple hardware regimes. However, both hardware implementations have restrictions in terms of transparency. First, using SMM-based introspection incurs high overhead (roughly 12μs to access one page), meaning that a malware sample could measure time elapsed during execution. Secondly, SlotScreamer potentially influences performance counters, which could be measured by stealthy malware with ring 0 privilege. Similarly, in systems where the PCI express bus is under high load, the use of SlotScreamer could adversely affect throughput.

VI. Related Work

There are two broad areas of work related to our technique. The first is debugging and analysis transparency. We focus on analysis techniques that reduce the presence of artifacts used to subvert malware analysis. The second is process introspection techniques. This body of work is focused on the collection and understanding of program memory during execution.

A. Malware Analysis and Debugging

Hops aims to aid analysis situations, such as heisenbugs or binal samples, that do not permit traditional debugging. Many techniques, including Ether [17], BitBlaze [42], and Anubis [4], provide debugging mechanisms to assist the analysis of malware. However, these techniques all rely on some manner of virtualization (or virtualization extensions like Intel VT). In contrast, we trade some confidence in analysis accuracy to achieve a higher level of transparency.

Several research projects have also begun to address the notion of transparency. V2E [49] combines Ether with more software emulation to replay malware execution, allowing an analyst to replay execution. However, V2E can be detected as it depends on virtualization. Additionally, its introduction of software emulation to support recording also causes significant slowdown—this adds to the problem of external timing attacks. SPIDER [16] implements invisible breakpoints using page table tricks. However, SPIDER depends on virtualization as well. Several studies have discovered these techniques [13], [35], [36], [40].

Aside from virtualization used in debugging, there are also explicit debuggers popular among malware analysts. For instance, IDA Pro [24] and OllyDbg [51] are popular debuggers that run alongside malware samples. DynamoRIO [10] uses process virtualization that executes on the OS and admits user-built dynamic instrumentation tools. These options require running software inside the target OS, which is easily detected by malware. In contrast, the proposed use of Hops does not add any software to the target system.

In recent years, Intel's System Management Mode (SMM) has appeared in the security literature. Spectre [52] uses SMM to introspect processes on live systems, successfully detecting malware. Similarly, MalT [53] uses SMM to provide interactive debugging remotely, though it did not address introspecting application-level software. SMM has also been used for system memory acquisition for forensic analysis [37], [47]. These techniques are quite similar to Hops in their ultimate goal. However, SMM-based techniques are not generalizable beyond x86 and suffer from external timing attacks due to significant performance overhead.

B. Process Introspection

Jain et al. [26] summarize numerous techniques for introspecting VMs and bridging the semantic gap, including process implanting [23] and process out-grafting [45]. Process implanting injects a process into the guest VM, relaying semantic information back to the hypervisor. This produces a potentially-obvious artifact for malware to observe (in addition to the VM framework itself). Process out-grafting attempts to address this problem by instead creating two VMs, one which runs the malware and one which runs an implanted process. This better prevents the implanted process from causing noticeable software artifacts at the expense of overhead.

Numerous projects have studied Virtual Machine Introspection [18], [22], [27], [29]. Such techniques focus on reconstructing semantic information about the guest kernel. This differs from Hops in that we are looking at a single process for the purpose of understanding potential malicious or stealthy behavior. Additionally, VMI techniques rely on virtualization, again introducing artifacts that can be used to subvert the analysis.

VII. Conclusions

Many software systems, from embedded devices to virtualization to security, cannot make use of standard debuggers. The act of analyzing a system can change that system, leading to heisenbugs in benign software and admitting anti-analysis by stealthy malware. We thus focus on zero-overhead approaches that leave no artifacts or traces that a program could use to behave differently when analyzed. We propose Hops, an approach to program introspection that infers and reports variable values and dynamic stack traces from hardware-provided memory snapshots. Our approach is based on two key observations. First, it is possible, using existing hardware, to log snapshots of memory pages with low to no overhead. Second, it is possible to bridge the semantic gap between raw memory snapshots and software semantics using a combination of program analysis, operating system, and security techniques.

Our approach formulates hypotheses about the locations of variables and stack frames, allowing analysts to introspect malicious and non-malicious programs. In our experiments, Hops was 84% accurate, overall, at reporting the values of local, stack-allocated, and global variables. We also implemented Hops using x86-based System Management Mode and PCIe-based SlotScreamer hardware to test our hypotheses on real hardware. In addition, it was over 50% accurate at reporting entire dynamic stack traces using conservative memory timings associated with available hardware. Third, in a human study involving 30 participants with statistical significance, Hops was no worse than `gdb` at supporting the analysis of standard maintenance questions. Finally, we examined 22 methods or artifacts that can be used by stealthy malware to detect analysis and observed that the introspection information provided by Hops was sufficient to reveal 16 of 22 artifacts used by pafish (and could reveal 5 more with faster hardware). Overall, we see Hops as an effective first step towards transparent process introspection.

VIII. Acknowledgments

This work was sponsored by the Assistance Secretary of Defense for Research and Engineering under Air Force Contracts #FA8721-05-C-0002 and #FA8750-15-2-0075 as well as the National Science Foundation under grants CCF0954024 and CCF1116289. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

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

[4] Anubis, "Analyzing unknown binaries," http://anubis.iseclab.org.

[5] M. Auty, A. Case, M. Cohen, B. Dolan-Gavitt, M. H. Ligh, J. Levy, and A. Walters. Volatility framework - volatile memory extraction utility framework. [Online]. Available: http://www.volatilityfoundation.org/

[6] E. Bachaalany, "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.

[7] S. Biedermann and J. Szefer, "Systemwall: An isolated firewall using hardware-based memory introspection," in *Information Security*. Springer, 2014, pp. 273–290.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

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

[10] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'12)*, 2012.

[11] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digital Investigation*, vol. 1, no. 1, pp. 50–60, 2004.

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

[13] X. Chen, J. Andersen, Z. Mao, M. Bailey, and J. Nazario, "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.

[14] Y. Chen, Y. Wang, Y. Ha, M. R. Felipe, S. Ren, and K. M. M. Aung, "saes: A high throughput and low latency secure cloud storage with pipelined dma based pcie interface," in *Field-Programmable Technology (FPT), 2013 International Conference on*. IEEE, 2013, pp. 374–377.

[15] DARPA, "Transparent Computing BAA," http://www.darpa.mil/Our_Work/I2O/Programs/Transparent_Computing.aspx, 2014.

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

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

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

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

[20] J. FitzPatrick and M. Crabill, "NSA Playset: PCIE," in *DEFCON 22*, 2014.

[21] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 177–187.

[22] Y. Fu and Z. Lin, "Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[23] Z. Gu, Z. Deng, D. Xu, and X. Jiang, "Process implanting: A new active introspection framework for virtualization," in *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 2011, pp. 147–156.

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

[25] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual." [Online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

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

[27] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through VMM-based out-of-the-box semantic view reconstruction," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

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

[30] C. Maartmann-Moe, "Inception," http://github.com/carmaa/inception, 2015.

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

[32] B. D. Payne. Libvmi: Simplified virtual machine introspection. [Online]. Available: https://github.com/bdpayne/libvmi

[33] C. E. Pitts, "Parallel processing support: so what is a "heisenbug" anyway?" in *Proceedings of the 17th Annual ACM SIGUCCS Conference on User Services, Bethesda, Maryland, USA, 1989*, 1989, pp. 237–242. [Online]. Available: http://doi.acm.org/10.1145/73760.73799

[34] M. Prandini and M. Ramilli, "Return-oriented programming," *Security & Privacy, IEEE*, vol. 10, no. 6, pp. 84–87, 2012.

[35] A. Quist and V. Smith, "Detecting the Presence of Virtual Machines Using the Local Data Table," http://www.offensivecomputing.net/.

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

[37] A. Reina, A. Fattori, A. Pagani, L. Cavallaro, and D. Bruschi, "When Hardware Meets Software: A Bulletproof Solution to Forensic Memory Acquisition," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC'12)*, 2012.

[38] S. P. Reiss, "Trace-based debugging," in *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, ser. AADEBUG '93. London, UK, UK: Springer-Verlag, 1993, pp. 305–314. [Online]. Available: http://dl.acm.org/citation.cfm?id=646902.710203

[39] K. Rosenfeld and R. Karri, "Attacks and defenses for JTAG," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 36–47, 2010. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/MDT.2010.9

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

[41] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 2006, pp. 23–34.

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

[43] T. I. Sookoor, T. W. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrodebugging: global views of distributed program execution," in *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems, SenSys 2009, Berkeley, California, USA, November 4-6, 2009*, 2009, pp. 141–154. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644053

[44] C. Spensky, H. Hu, and K. Leach., "LO-PHI: Low Observable Physical Host Instrumentation," in *Proceedings of 2016 Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[45] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, “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 (CCS’11)*, 2011.

[46] J. Stüttgen and M. Cohen, “Anti-forensic resilient memory acquisition,” *Digital Investigation*, vol. 10, pp. S105–S115, 2013.

[47] J. Wang, F. Zhang, K. Sun, and A. Stavrou, “Firmware-assisted memory acquisition and analysis tools for digital forensic,” in *Proceedings of the 6th International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE ’11)*, 2011.

[48] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, “Drdebug: Deterministic replay based cyclic debugging with dynamic slicing,” in *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO ’14, Orlando, FL, USA, February 15-19, 2014*, 2014, p. 98. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544152

[49] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin, “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. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151053

[50] P. Yu, L. Bo, L. Datong, and P. Xiyuan, “A high speed dma transaction method for pci express devices,” in *Testing and Diagnosis, 2009. ICTD 2009. IEEE Circuits and Systems International Conference on*. IEEE, 2009, pp. 1–4.

[51] O. Yuschuk, “OllyDbg,” www.ollydbg.de.

[52] F. Zhang, K. Leach, K. Sun, and A. Stavrou, “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.

[53] F. Zhang, K. Leach, H. Wang, A. Stavrou, and K. Sun, “Using hardware features for increased debugging transparency,” in *Proceedings of the 36th IEEE Symposium on Security and Privacy*, 2015.