PIEtrace: Platform Independent Executable Trace

Yonghwi Kwon, Xiangyu Zhang and Dongyan Xu
Department of Computer Science, Purdue University
West Lafayette, Indiana, 47907
Email: {kwon58,xyzhang,dxu}@cs.purdue.edu

***Abstract*—To improve software dependability, a large number of software engineering tools have been developed over years. Many of them are difficult to apply in practice because their system and library requirements are incompatible with those of the subject software. We propose a technique called *platform independent executable trace*. Our technique traces and virtualizes a regular program execution that is platform dependent, and generates a stand-alone program called the *trace program*. Running the trace program re-generates the original execution. More importantly, trace program execution is completely independent of the underlying operating system and libraries such that it can be compiled and executed on arbitrary platforms. As such, it can be analyzed by a third party tool on a platform preferred by the tool. We have implemented the technique on x86 and sensor platforms. We show that buggy executions of 10 real-world Windows and sensor applications can be traced and virtualized, and later analyzed by existing Linux tools. We also demonstrate how the technique can be used in cross-platform malware analysis.**

I. Introduction

To improve software dependability, researchers and engineers have developed a large number of software engineering tools over years. Although many of them have very advanced capabilities, they do not gain popularity in practice. One of the issues is that they often require certain environmental support, such as specific operating system, libraries, and the presence of some infrastructure (e.g. Valgrind). These requirements may be incompatible with those of the subject software. For instance, Windows software cannot make use of the large body of Linux tools. A subject software that requires a new version of `libc` can hardly use a runtime tool based on an older version of `libc`. In our personal conversation with researchers and engineers from Coverity®, one of the most successful software testing and debugging service providers, it was mentioned that making Coverity's tools run on the customers' machines and their software is one of the most prominent challenges. They indeed have a team of developers whose responsibilities are solely in ensuring compatibility. The team is even larger than most of their tool development teams.

Compatibility and platform independence are not a new challenge in our discipline. In fact, people have made a lot of progress in mitigating such problems in recent years. The invention of virtual machines allows running different guest operating systems on a singe host operating system. However, virtualization is performed at the machine level and hence applying a cross-platform software engineering tool is still difficult. For example, one cannot apply an existing Linux tool on a Windows program execution even with the help of VM techniques. We also have excellent cross-platform analysis infrastructures such as Pin that support both Linux and Windows such that a Pin tool developed for Linux programs only needs small modifications to adapt to Windows programs. However, only tools developed on Pin can benefit from this feature. Moreover, Pin only supports a very limited set of instruction sets. For example, one cannot use Pin tools to analyze sensor program execution. Programs and tools written in languages such as Java are by their nature platform-independent due to the presence of Java Virtual Machine. However, there are many programs that run natively without any virtual machine support. PIEtrace is focused on those programs.

Modern compiler infrastructures, e.g. LLVM, also try hard to improve portability and mitigate cross-platform issues. They can compile code written in different languages and for different platforms to the same intermediate representation (IR) such that *static* program analysis written on these infrastructures can be used to analyze software from different platforms. However, it is only good for static analysis. The execution of a program compiled by these infrastructures is nonetheless platform dependent. In other words, these infrastructures offer limited help in cross-platform application of dynamic analysis tools.

In this paper, we propose a novel technique called *platform independent executable trace* (PIEtrace). Our technique traces and virtualizes a regular program execution (or part of it) that is platform dependent, and generates a program called the *trace program*. A trace program is a stand-alone program that can be compiled and executed on other platforms. Running a trace program re-generates the original execution. More precisely, *it reproduces an execution that exhibits the same user-space control flow, values and program dependences as the original execution*. Trace program execution is completely independent of the underlying operating system and any library such that it can be on arbitrary platforms. The application scenario is as follows. We provide a tracing/virtualization component for each platform that we support. Such components are written in infrastructures such as Pin and sensor emulators. Executing a program on the component produces the corresponding trace program. The trace program can be compiled and executed on any environment that a third party tool prefers. For example, one can compile and execute a trace program generated from a sensor program execution on a Linux platform such that Linux runtime tools can be applied. Our technique also supports projecting the analysis results back to the original platform, to facilitate in-context human inspection.

Compared to the traditional tracing techniques [24], [46], we do not require tools to support a specific trace format. Existing third party tools on various platforms that analyze program execution can be used to analyze trace program execution. This would greatly improve tool applicability. Moreover, our technique does not require the physical presence of trace, which is very resource consuming. Compared to traditional logging and replay techniques [6], [33], trace program execution can be

cross-platform (even cross-instruction set), without requiring the original OS or libraries. Trace programs execute on their own, without any replay runtime support.

Our contributions are summarized as follows.

- We propose the novel concept of *platform independent executable trace*, which can be considered as an adaptor for existing dynamic analysis tools.
- We address the underlying technical challenges. Trace program is not a simple recording of the sequence of executed instructions. It is more like a transformation of the original program. An instruction in the original program has only one virtualized copy in the trace program even though it may be executed many times. Our technique suppresses dependences on the underlying hardware, operating systems, and libraries, leveraging a novel lazy-logging method. The method does not require the tracing component to understand any system/library interfaces to achieve independence. Our technique ensures that execution of the trace program generates the original control flow by handling direct, indirect, and even unexpected control flow transfers caused by interrupts and exceptions. It also guarantees the same values and data flow to be reproduced by properly virtualizing memory and registers of the original program.
- We have implemented PIEtrace on x86 and sensor platforms. We evaluate PIEtrace by tracing/virtualizing buggy executions of 10 real-world Windows and sensor applications including `Acrobat Reader`. We then use existing Linux debugging tools to identify the root causes of these bugs. We also show case that our technique can be used to virtualize Windows malware execution including packed malware so that the malware trace programs can be executed and analyzed as many times as desired. PIEtrace and its benchmarks are available at [23]

Limitations. PIEtrace guarantees to reproduce the user space behavior of the original execution, including control flow, data flow and values. Library function execution is in the user space and hence faithfully reproduced. In contrast, system level behavior, such as the execution inside a system call made by the original program is invisible to PIEtrace. PIEtrace does not know the semantics of system calls either, as it does not need to. However, it faithfully records the system call invocations (including system call numbers). As such, some malware behavior analysis that relies on interpreting the *values* passed into and acquired from the kernel through specific system calls may not be applicable. PIEtrace has limited support for multi-threaded executions. It generates a trace program for each thread. Although it faithfully captures individual thread executions, analyzing thread interleaving requires additional work. PIEtrace currently has non-trivial runtime overhead. It is not intended to be deployed in a production setting.

II. Demonstrative Example

In this section, we use a Windows malware example to demonstrate our technique. It is a Browser Helper Object(BHO) malware[1] executing on Internet Explorer(IE). BHO is a plug-in of IE that can register callbacks for important events. The malware monitors all visited URL so that an advertisement is displayed when some particular URLs are detected. In this case study, we want to identify and understand the malicious payload of the BHO plug-in.

Firstly, we observe that the DLL binary file of the plug-in has a non-trivial size (300kb) and complex structure. The left side of Fig. 1 shows the static call graph generated by IDA, the most popular binary analysis tool used in malware analysis. The graph has 222 nodes and 1330 edges. It is difficult for a human to determine the malicious payload inside the complex graph. Besides, the graph is incomplete, missing many call edges through function pointers due to the difficulty of static analysis. The plug-in has a lot of benign functions for initialization and normal processing.

An alternative is to observe the plug-in execution. However, the plug-in has to execute as part of IE. That implies a tool has to load and monitor the IE execution, which is highly complex and expensive to monitor. In an execution in which we load 3 pages with one of them triggering the advertisement, the execution of the IE process is 151 times larger than the sub-execution of the plug-in. Furthermore, to the best of our knowledge, there is not an existing scalable and publically available Windows dynamic analysis tool that we can easily leverage to understand the malware.

Therefore, in this case study, we use PIEtrace to trace and virtualize the execution of the plug-in. We then use a Valgrind based tool Callgrind on Linux to generate the dynamic call graph of the trace program, whose execution regenerates the original plug-in execution. To further determine the malicious payload, we use a simple approach to prune the benign behavior in the trace program. Particularly, we trace and virtualize two IE executions, one accessing the full set of pages including the one that could trigger the advertisement and the other accessing the two benign pages only. We acquire the two trace programs with sizes of 453KB and 1.7MB. We execute them on Callgrind and generate two call graphs for comparison. The right hand side of Fig. 1 shows the result. The shaded nodes are the functions that uniquely appear in the execution with the advertisement. There are totally 16 such functions and we suspect they are the malicious payload. Since our trace program contains information of calls to external APIs, we are able to understand the behavior of some of the functions by observing the external API calls in these functions.

In particular, the malware first checks if there are other loaded BHOs. If so, it unloads them in order to gain exclusive control of the browser(in node A). Then, it shows an advertisement dialog on the screen (in node B) and stops the current navigation of the browser(in node C). While the behavior of (B) and (C) corresponds to the observed symptoms of the malware, the graph also reveals its stealth behavior (A). It calls `CConnectionPoint::EnumConnection` in `ieframe.dll`. Although this interface is supposed to be called by the browser to manage registered BHOs, the malicious BHO uses it to disable other BHOs.

[1]Analysis record of the main executable of malware can be found at https://www.virustotal.com/file/5900e4073c57b2246724b581eba9eed76787fb54b38f7119d301f6713856ee0a/analysis/

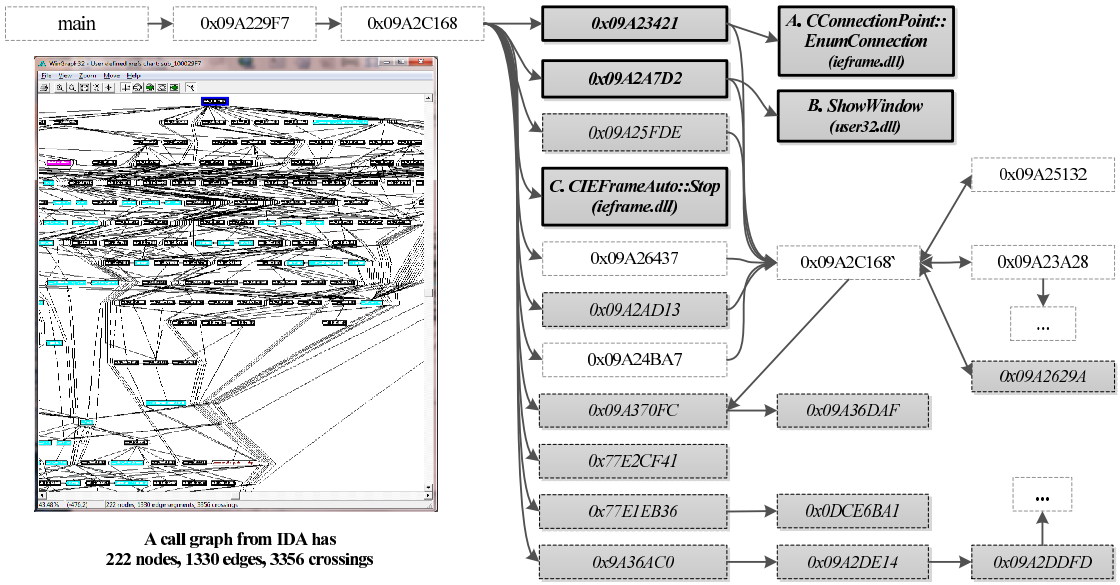

Fig. 1. BHO call graphs. The one on the right is generated by our analysis. Shaded nodes denote malicious behavior.

III. Basic Design

Program	P	$::= s$
Stmt	s	$::= s_1; s_2 \mid \textbf{skip} \mid r :=^{\ell} e \mid r :=^{\ell} \textbf{R}(r_a) \mid \textbf{W}^{\ell}(r_a, r_v) \mid \textbf{goto}^{\ell}(\ell_1) \mid \textbf{if } (r^{\ell}) \textbf{ then goto}(\ell_1) \mid \textbf{syscall}^{\ell}() \mid \textbf{depinst}^{\ell}() \mid r := \textbf{malloc}^{\ell}(r_s) \mid \textbf{free}^{\ell}(r) \mid \textbf{call}^{\ell}(\ell_1) \mid \textbf{ret}^{\ell}$
Operator	op	$::= + \mid - \mid * \mid / \mid ...$
Expr	e	$::= r \mid c \mid a \mid r_1 \textbf{ op } r_2 \mid r \textbf{ op } c$
Register	r	$::= \{sp, r_1, r_2, r_3, ...\}$
Const	c	$::= \{true, false, 0, 1, 2, ...\}$
Addr	a	$::= \{0, 1, 2, ...\}$
Label	ℓ	$::= \{\ell_1, \ell_2, \ell_3, ...\}$

Fig. 2. Language

Language. To facilitate discussion, we introduce a low level language to model binary executables. For simplicity, we only model enough to illustrate the key ideas. The language and the discussion are general, not bounded to a specific instruction set. The syntax is presented in Fig. 2.

Memory reads and writes are modeled by $\textbf{R}(r_a)$ and $\textbf{W}(r_a, r_v)$ with r_a holding the address and r_v the value. Since it is a low level language, we do not model conditional or loop statements, but rather jumps using **goto** and guarded **goto**; **syscall**() denotes system calls[2]; **depinst**() represents instructions that depend on the specific platform; **malloc**(r) and **free**(r) represent heap allocation and deallocation. Function invocations and returns are modeled by **call**(ℓ) and **ret**. Note that we use sp to denote the stack pointer register and we assume all platforms have such a register.

Our technique executes the original program P. During execution, it constructs another program P' that is platform-independent, not depending on any OS, libraries, input files, or network connections. Running P' regenerates the original execution. *More precisely, it regenerates an execution that has the same user-space control flow, data flow, and values as the original execution*. To simplify our discussion, one can consider P' a program in a sub-language of the one in Fig. 2. In particular, P' will not have any system call instructions, platform specific instructions, heap memory management instructions, or registers except the stack register.

Next, we elaborate the technique from different aspects.

A. Control Flow Tracing and Virtualization

The first aspect focuses on ensuring the trace program will regenerate the same control flow when it executes. For each executed instruction in the original program, we record it in a buffer. For control transfer instructions (e.g. jumps and calls), we patch the control transfer target to ensure correct control flow. If an instruction gets executed multiple times, our technique keeps only one copy of its virtualized version to minimize the size of the trace program.

Upon execution of an instruction, the technique checks if it is executed for the first time. If so, it is stored in the buffer. At the end, the buffer is the trace program. In the tracing/virtualization rules in Table I, function **AddStmt**(ℓ, s) (defined in Fig. 3) is called upon execution of any instruction (with label ℓ) to record the instruction. Parameter s denotes the virtualized version of the instruction, which may contain more than a few instructions to achieve platform independence. We use a *VirtualLabelMap* VL (Fig. 3) to map an instruction in the original executable to its virtualized version in the trace program. Hence inside **AddStmt**(), we first test if the instruction has been traced before by checking VL. If not, the instruction is attached to the end of the trace program and the mapping is updated.

For jump and call instructions (rules JUMP and CALL), we look up the new label of the jump/call target using function **GetLabel**(ℓ) and redirect the control to the new position. Note that when we call **GetLabel**, the target instruction may not be executed and virtualized yet. In this case, the function returns the end position of the trace buffer, at which the target instruction will be put.

We also take special care of call and return instructions. In particular, we cannot simply use call and return instructions in

[2]We do not model their parameters as our technique does not require such information.

$P' \in Program$ ::= $\overline{s}$
$VL \in VirtualLabelMap$ ::= $Label \rightarrow Label$
$VA \in VirtualAddrMap$ ::= $Addr \rightarrow Addr$
$M \in MemStore$ ::= $Addr \rightarrow Const$
$REG \in RegStore$ ::= $Register \rightarrow Const$
$SM \in ShadowMemory$ ::= $Addr \rightarrow Const$
$ACC \in Accessed$ ::= $Addr \rightarrow Boolean$
$LS \in LogStorage$ ::= $LogId \rightarrow LogEntry$
$id \in LogId$ ::= $< \ell, z >$
$E \in LogEntry$ ::= **MEMLOG**(a,c) | **REGLOG**$(c_{sp}, c_1, c_2, c_3, ...c_n)$
$HC \in HitCount$ ::= $Label \rightarrow \mathbb{Z}$
$VR(r)$ represents the global var. in the trace program representing r

GetLabel(ℓ) ::=
  if $VL(\ell)$ *is empty* **then**
    return the end position of P';
  else
    return $VL(\ell)$;

AddStmt$(\ell,\ s)$ ::=
  if $VL(\ell)$ *is empty* **then**
    $VL(\ell) :=$ *the end position of* P';
    $P' := P' \cdot s$;

LogRead$(\ell,\ a)$ ::=
  $HC(\ell)++$;
  if $a \neq null$ **then**
    if $SM(a) \neq M(a)$ **then**
      $LS(< \ell, HC(\ell) >) :=$ **MEMLOG**$(a,\ M(a))$;
    $SM(a) := M(a)$;

Concretize(ℓ) ::=
  $HC(\ell)++$;
  $t := LS(< \ell, HC(\ell) >)$
  if $t \equiv$ **MEMLOG**(a,c) **then**
    W$(VA(a), c)$
  if $t \equiv$ **REGLOG**$(c_{sp}, c_1, ..., c_n)$ **then**
    $VR(sp) := c_{sp}; ...; VR(r_n) := c_n$

Fig. 3. Definitions for virtualization rules

the trace program because they entail implicit stack operations. Upon a function call, the return address (i.e. the address of the instruction immediately following the call instruction) is *implicitly* pushed to the stack. Upon return, the return address is loaded from the stack. If we use a call instruction in P' to virtualize a call in P, the address pushed to the stack is the neighboring instruction in P', which does not necessarily correspond to the instruction immediately succeeding the call instruction in P, leading to incorrect control flow upon return. Therefore, as shown in rule CALL, our solution is to explicitly push the *original* return address, i.e. $\ell+1$, and then use a jump instruction to virtualize the call. Upon return (rule RET), the original return address is read and then translated *at runtime* using VL.

B. Data Flow Tracing and Virtualization

In the second aspect, we discuss how to ensure execution of the trace program reproduces the same data flow in a platform-independent fashion. This is achieved by virtualizing memory layout, memory management, and registers, which are platform dependent. For example, the typical stack address of Linux applications is not available in Windows because it is reserved for the Windows kernel. To virtualize memory, we track all the memory locations and regions accessed in the original execution and then declare these locations and regions as *global* buffers in the trace program. The original accesses are redirected to these buffers. One critical property we want to ensure is that all the original user space memory accesses, including their addresses and the values accessed, can be preserved by our virtualization. This is critical for cross-platform debugging as unsafe accesses such as buffer overflows are preserved too. Registers need to be virtualized too as different platforms support different registers. We use a set of global variables to represent registers.

We use a mapping ACC that maps an address to a boolean value to denote if the address has ever been accessed during execution. Hence, in rules READ, WRITE, CALL, the mapping is set. In a call instruction, the stack memory is implicitly accessed to store the return address. Observe in these rules, $VR(r)$ denotes the global variable in the trace program that is used to represent register r; REG is the register store and hence $REG(r)$ denotes the current value in r.

After the original execution terminates, we process the ACC mapping to divide all the accessed addresses to a number of regions. Ideally, we can allocate one variable in the trace program for each unique accessed address. However, this requires maintaining a very large address translation table to map an original address to its corresponding variable in the trace program. On the other hand, we can allocate a large buffer to denote the entire virtual space in the original execution. In such a case, although we only need to perform linear address translation, the space consumption is large. Hence, we divide the accessed addresses to a few regions to achieve a good tradeoff between the memory overhead and the address translation overhead. In particular, *we consider any two addresses with a distance less than a predefined threshold (256 in this case) belong to the same region*. For each identified region, we declare a global buffer in the trace program. During execution of the trace program, a memory address is translated on the fly right before it is accessed, by first performing a range query to determine the buffer for its region and then offsetting to the right location.

In rule READ, the second instruction added by **AddStmt**() translates the address to be read $VR(r_a)$, which is an address in the *original* execution, to a by calling $VA()$, which performs a range query and then offsetting. Its details are elided. Rule WRITE is similar.

Doing so, our scheme does not rely on any platform specific memory layout or memory management policy. Intuitively, trace programs do not allocate any heap memory as heap allocation and deallocation functions are platform dependent. The original stack manipulations are virtualized and emulated on a global region.

Example. Assume $0x08000400$, $0x08000432$, $0x080004b0$ and $0x8000900$ are accessed. We determine the first three form a region and the last forms another region. We allocate a buffer A with the size ($0x80004b0$+4-$0x8000400$) for the first region. Assume during execution of the trace program $0x08000432$ is about to be accessed, the translation identifies that A is its buffer as it falls in the range of $[0x08000400, 0x080004b4]$. Its actual address to access is hence $\&A + (0x8000432 - 0x08000400)$. □

In our design, the entire stack footprint (i.e. the maximum stack consumption) of the original execution is often determined as a large region, as stack accesses tend to be close to each other. The heap is often divided into smaller regions.

Statement	Action	Rule
$r :=^{\ell} \mathbf{R}(r_a)$	$ACC(REG(r_a)) = true$; **LogRead**$(\ell,\ REG(r_a))$; **AddStmt**$(\ell,$ "**Concretize**(ℓ); $a := VA(VR(r_a))$; $VR(r) := \mathbf{R}(a)$;");	READ
$\mathbf{W}^{\ell}(r_a, r_v)$	$ACC(REG(r_a)) = true$; **AddStmt**$(\ell,$ "$a := VA(VR(r_a))$; $\mathbf{W}(a, VR(r_v))$;");	WRITE
goto$^{\ell}(\ell_1)$	$\ell' :=$ **GetLabel**(ℓ_1); **AddStmt**$(\ell,$ "**goto**(ℓ');");	JUMP
if(r^{ℓ}) **then** **goto**(ℓ_1)	**AddStmt**$(\ell,$ "**if** $(VR(r))$ **then goto**$(VL(\ell_1))$;");	COND-JUMP
syscall$^{\ell}()$ | **depinst**$^{\ell}()$	$LS(<\ell, ++HC(\ell)>) :=$ **REGLOG**$(REG(sp), REG(r_1), ..., REG(r_n))$; **AddStmt**$(\ell,$ "**Concretize**(ℓ);");	SYSDEP
malloc$^{\ell}(r)$ | **free**$^{\ell}(r)$	*skip*	HEAP
call$^{\ell}(\ell_1)$	$ACC(REG(sp)-1) = true$; $\ell' :=$ **GetLabel**(ℓ_1); **AddStmt**$(\ell,$ "$sp' := sp' - 1$; $\mathbf{W}(sp', \ell+1)$; **goto**(ℓ');");	CALL
ret$^{\ell}$	**AddStmt**$(\ell,$ "$t := \mathbf{R}(sp'++)$; **goto**$(VL(t))$;");	RET

Register sp' is the stack pointer register in the trace program. Note that the actions in the second column are performed during the original execution, whereas the statements added by **AddStmt**() constitute the trace program and will be executed separately.

TABLE I. TRACING/VIRTUALIZATION RULES.

A region identified by our analysis may not correspond to an allocated heap region. However, this does not affect the soundness of our technique.

Theorem 1. *All user space memory accesses, including buffer overflows, in the original execution are preserved. Particularly, the same sequence of addresses and accessed values is regenerated by trace program execution.*

Intuitively, all addresses are maintained and manipulated in their *original* form during execution of the trace program. They are only translated right before the access. Hence, assume in the original run a buffer A is overflowed and thus its neighboring buffer B is overwritten. The write to the address within buffer B is faithfully reproduced as the same faulty pointer manipulation that overflows A will be faithfully replayed. It is independent of the region identification in our virtualization process. Note that some memory related exceptions such as null pointer dereferences will explicitly terminate the tracing process, they are essentially preserved by the termination of the trace program. The formal proof is omitted.

Note that maintaining identical original addresses during trace program execution is critical, even though they are not accessible. That is because some functions take different execution paths according to addresses. For example, text formatting functions such as `printf()` and `fprintf()` execute different parts of the function according to the input pointer values. Also, the execution path of `EncodePointer()` and `DecodePointer()` in Windows depends on the given address. Furthermore, there are instructions that require memory alignment. In x86 architecture, SSE and MMX instructions such as MOVAPS and MOVNTPS are examples. To execute these instructions properly, we align all the base addresses of virtualized memory regions according to the original base addresses. Since the least common multiple (LCM) of alignments of SSE and MMX is 16-byte, PIEtrace also applies 16-byte alignment to all virtualized memory regions.

C. System-level Dependence Elimination

A real execution is most likely system dependent. For instance, it may have to perform system specific I/O by system calls (e.g. read from a specific device); it may execute system-dependent instructions such as CPUID. We have to eliminate such dependences in order to run the trace program on a different platform. Note that libraries are not a problem for PIEtrace as we are able to trace into library execution. In other words, library execution will be part of the trace program. We also need to handle non-deterministic instructions such as RDTSC (read current timestamp). Otherwise, they will cause execution differences. Currently, we have limited support for concurrency. PIEtrace generates a trace program for each thread. While trace programs capture the user-space behavior of individual threads, PIEtrace does not currently support reasoning about thread interleavings.

A typical solution to handle system calls by many existing logging and replay tools [33], [38], [2], [6] is to record the values read/written by system calls (e.g. the packet received by a socket read). During replay, instead of interacting with the real device, it simply restores the recorded values from the log. Despite its simplicity, such a design is platform-dependent because the logging and replay has to be aware of the entire system call interface, such as which part of memory is being updated during a socket read.

We develop a new solution. Instead of directly tracking system calls or platform dependent instructions. We develop a *lazy-logging* approach. During tracing, we maintain a shadow memory that can be considered as parallel to the normal memory. When a user-space instruction of the original program updates the memory, which is visible to our tracing system, we update the shadow memory in the same way. When a kernel-space instruction (inside a system call) updates the memory, the shadow memory cannot be updated because such writes are invisible to PIEtrace. Later, when the memory updated by the system call is read, the discrepancy between the normal memory and the shadow memory suggests the previous invisible update. We then log the value in the memory. It will be restored during trace program execution. Besides system calls, the method also naturally handles other platform-dependent instructions and non-deterministic instructions including remote thread reads and writes.

As shown by rule READ in Table I, function **LogRead**() is called with the address. The function is defined in Fig. 3. It first updates the hit count of the instruction, which counts the number of instances of the instruction. It then compares

the shadow memory and the actual memory. If they differ, a log entry identified by the instruction label and its hit count is added. The hit count is to handle the case that a read instruction gets executed many times and only some instances have their values updated by system calls. Others have their values updated by user-space instructions and hence can be re-computed during the trace program execution. They do not need to be logged.

As part of the rule, a call to function **Concretize**() is added to the trace program, which will be executed when the trace is replayed. Symmetric to **LogRead**(), it also first increases the hit count, and then it checks if there is a log entry associated with the current instance identified by the label and the hit count. If so, it sets the corresponding memory or registers to the recorded values.

In rule SYSDEP, even though we do not need to log any values in memory due to the lazy logging strategy, we do log the current register values. The reason is that register values may be changed by system calls or platform dependent instructions and such changes need to be captured.

IV. IMPLEMENTATION

To make the technique practical, we also need to address a number of implementation challenges.

Handling Indirect Control Flow Transfer and Long/Set Jumps. Indirect control flow transfer and long/set jumps are commonly used in x86 executables. Indirect control flow transfers are used to support function pointers, virtual function calls in object oriented programs, and jump tables compiled from `switch-case` statements. Long/set jumps are usually used to implement control transfer to exception handlers. They share the same characteristics that the control transfer target of a jump/call instruction is a runtime value. For example, they may be in the form of "**goto**(r)" with r the register holding the dynamic target. Hence, the control transfer rules JUMP, COND-JUMP, and CALL in Table I are insufficient as their targets are a constant program label. Our solution is to look up the new target from the virtual label map VL during the trace program execution. For example, we have the following rule for the indirect jump statements.

Statement	Action
$\textbf{goto}^{\ell}(r)$	**AddStmt**(ℓ, "**goto**($VL(VR(r))$;");"

Indirect calls are similarly handled. Note that our technique guarantees values generated in the virtualized registers during trace program execution, e.g. in $VR(r)$, are identical to those generated in the real registers in the original execution. This guarantees the correctness of the above indirect jump rule. The use of the VL map in the trace program also implies that we need to provide it as part of the trace program. We declare it as a global array.

Unexpected Control Flow. Unexpected control flow is very common in modern systems. Windows/Linux use unexpected control flow to support hardware/software interrupts and exception handling. Real-time/embedded systems are even largely driven by interrupts. Upon a hardware/software interrupt, execution is trapped to the kernel mode, which is invisible to our tracing system. The kernel looks up the pre-defined interrupt table to determine which user-mode function needs to be called based on the interrupt or some pending hardware signal. Similarly, if an exception happens, the execution is trapped to the kernel, which determines the handler to call. Note that hardware interrupts and exceptions could theoretically happen at any instruction. Hence, in the user mode, we may observe unexpected control flow from any instruction to some interrupt/exception handler. Such changes can hardly be predicted.

To handle this issue, after execution of an instruction, we predict the address of the next instruction, denoted by $InstAddr_p$, based on the current instruction $InstAddr_c$. If the current instruction is not a branch, $InstAddr_p$ is "$InstAddr_c$ + size($InstSize_c$)". For a branching instruction, if the branch is taken, $InstAddr_p$ is the branch target, otherwise the fall-through. When the tracing system sees the real next (user-mode) instruction, we check if our prediction is correct. If not, our system detects an unexpected control flow change. Hence, we log the values of all registers, including the program counter, which denotes the current instruction address. During trace program execution, they are restored after the execution of $InstAddr_c$. Note that restoring the program counter register automatically replays the unexpected control transfer. We need to log other register values as well because they may change due to the invisible kernel mode execution. The detailed semantic rule is omitted due to the space limitations.

Symbolic Information Preservation. An important design goal is to allow the trace program to be analyzed by different tools on various platforms. However, we want to interpret the analysis results on the original system. As such, we need to preserve the symbolic information of the original program in the trace program. Symbolic information of an executable often includes the source code file and line for each instruction, variable type and its declaration file, and so on. In particular, during the tracing/virtualizing process, we generate an offline dictionary. It maps each instruction in the trace program to an instruction in the original program, whose symbolic information can be looked up from the original executable. For each variable accessed in the original run (as captured by *ACC*), we also preserve the mapping from its buffer address in the trace program to the variable.

Supporting Different Instruction Sets. PIEtrace supports cross-instruction-set trace virtualization and replay. Besides x86, our implementation can also trace Mica2 sensor program execution and generate trace programs in x86 so that the wealthy set of x86 tools can be used to analyze sensor program execution. The additional challenge lies in eliminating the instruction set differences. For Mica2 instructions that do not have direct correspondence in x86, for example SBIC, we provide functions written in x86 to emulate them.

V. EVALUATION

We evaluate PIEtrace with real world applications on x86 and sensor platforms. For x86, we run 5 real world buggy programs on Windows and collected the trace programs of the faulty runs for cross-platform analysis. The five Windows programs are downloaded from a vulnerability database [35]. Note that although the database lists many vulnerabilities, most of them do not provide the exploit inputs or the exploits cannot be reproduced. After we randomly inspected a large set of the listed vulnerabilities of COTS software, we found these five that have the inputs available and can

Platform	Subject Software	# of Inst. (Dynamic/Static)	Program size (Original[a]/Trace-program)	Log size (Plaintext/Compressed[b])	Bug
x86	Acrobat Reader 9.3	208M / 506K	342KB / 31MB	78MB / 31MB	Memory corruption
x86	CastRipper	103M / 230K	564KB / 13MB	25MB / 4MB	Buffer overflow
x86	Microsoft HTML Help	7M / 117K	11 KB / 6MB	4MB / 1MB	Buffer overflow
x86	PowerTabEditor	37M / 186K	2.16 MB / 10MB	8MB / 2MB	SEH/EIP corruption
x86	FreeAmp	67M/ 120K	272 KB / 6MB	16MB / 4MB	Buffer overflow
ATMega128[c]	Sensor node by [17]	8M / 2K	31KB / 204KB	2MB / 1MB	Buffer overflow
ATMega128	MultihopOscilloscope [25]	7 M / 2K	54KB / 214KB	2MB / 1MB	Data race
ATMega128	CntToLedRfm [44]	3 M / 1K	18KB / 155KB	700KB / 57KB	Concurrency bug
ATMega128	BlinkFail [39]	32K / 461	6KB / 78KB	22KB / 3KB	Memory safety
ATMega128	RfmToLed [30]	679K / 1K	17KB / 154KB	923KB / 75KB	Zero length packets
x86	Packed-Malware	246M / 239K	887 KB / 19MB	22MB / 8MB	
x86	Multi-Packed Malware	322M / 263K	977 KB / 20MB	47MB / 12MB	
x86	BHO	4M / 33K	264 KB / 1MB	312KB / 123KB	

a. application binary only, not including dynamic libraries; **b.** 7zip utility was used; **c.** ATMega128 is a sensor instruction set.

TABLE II. BENCHMARKS AND VIRTUALIZATION RESULTS.

Name	Tested extensions or functionalities
Pin	Program slicing tool and Memtrace
Valgrind	Memcheck+ and Callgrind
DynamoRIO	Instrcalls
OllyDebugger	WatchMan, OllyScript, HitTrace
Immunity Debugger	FullDisasm and Ariadne
Windbg	PyDbgEng and Windbg Script
gdb	Reverse debugging (reverse-step, reverse-continue)

TABLE III. EXISTING ANALYSIS TOOLS TESTED ON OUR SYSTEM.

be reproduced. We execute the subject programs with the provided vulnerable inputs to get the trace programs. In the five applications, `CastRipper` is a recording program and `Microsoft HTML Help` is the default HTML help file loader. `PowerTabEditor` is a guitar note editor program and `FreeAmp` is a music player.

For the sensor platform, we run 5 buggy sensor applications on an emulator called ATEMU [31], The supported instruction set is `ATMega128`, a kind of RISC instruction set. The operating system is TinyOS. The five sensor buggy applications are mainly collected from existing literature [17], [25], [44].

We also evaluate PIEtrace on three malwares to show that we can effectively help malware analysis, even when a malware is packed. The IE-BHO case discussed in Section II is one of them.

The set of subject programs and the corresponding bugs are presented in the second column and the last column of Table II, respectively. PIEtrace and the test benchmarks are publically available at [23]. We also run PIEtrace on SPECINT 2000 programs to study overhead. These programs are bug free.

A. Virtualization Results

The virtualization results can be found in Table II. The third column shows the number of executed instructions (static) and their instances (dynamic). The static number is also the number of instructions that get virtualized. Column four shows the size of programs including the original and the corresponding trace programs. Observe that the trace programs are usually much larger than the original programs. That is because an instruction is usually virtualized to a few instructions and a trace program includes all the libraries used including those dynamically loaded. Furthermore, we have to include the virtual label mapping VL and the virtual address mapping VA (defined in Fig. 3) as part of the trace programs. However, our later experiment will show that the trace program size does not change much over time.

Column five shows the log size. More detailed results about the instructions that trigger logging and their effects are presented in Table IV. The "Data" columns present the numbers of instruction instances (dynamic) and unique instructions (static) that need logging because of invisible system level memory writes. Their percentages (over the total number of dynamic and static instructions, respectively) are also presented. The "Control Flow Change" columns show logging caused by unexpected control flow changes. The "Platform-Dependent Inst." columns present logging for platform dependent instructions (e.g. CPUID). The last two columns show the total. Observe that for most cases, only a small percentage of all executed instructions triggers logging except for two sensor cases. That is because those two cases are very I/O bound. For Windows cases, most loggings are caused by data differences whereas for sensor cases, most are caused by platform-dependent instructions. That is because sensors use the platform-dependent instructions IN and OUT to perform one byte hardware read and write. Another observation is that unexpected control flow happens very often at a very small number of places. That is due to the event driven execution model (i.e. program execution is trapped to kernel through interrupt instructions and various user mode handlers may get called depending on the events).

B. Cross-Platform Analysis

An important goal of PIEtrace is to enable cross-platform dynamic analysis, namely, using a tool on a specific platform to analyze an execution on a different platform. In this experiment, we apply a set of tools as shown in Table III to the trace program executions. The first column shows the infrastructures of the tools. The second column shows the tools, which are usually implemented as infrastructure extenstions. Pin (Linux x86), Valgrind (Linux x86), and DynamoRIO (Windows) are dynamic binary instrumentation engines. OllyDebugger, Immunity Debugger, Windbg, and gdb are debuggers that are widely used on Windows or Linux.

We applied two Pin tools, namely, a dynamic slicing tool [21] and the Memtrace tool, to the trace program executions. The slicer detects both data and control dependences during execution and performs backward/forward slicing given a slicing criterion. Since PIEtrace preserves user-space dependences, the slicer produces the same slices as the ones generated when it is applied to the original executions. Note that the slices contain some instructions that are for the purpose of

Subject Software	Data		Control Flow Change		Platform-Dependent Inst.		Total	
	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static
Acrobat Reader 9.3	3M(1.8%)	9K(1.9%)	19K[a]	3[a]	21K[a]	9K(1.9%)	3M(1.8%)	9K(1.9%)
CastRipper	703K[a]	6K(3%)	147K[a]	3[a]	147K[a]	6K(3%)	998K(1%)	6K(3%)
MS HTML Help	226K(3.1%)	3K(3.4%)	952[a]	4[a]	1K[a]	3K(3.4%)	229K(3.1%)	3K(3.4%)
PowerTabEditor	261K[a]	6K(3.2%)	36K[a]	3[a]	36K[a]	6K(3.2%)	335K[a]	6K(3.2%)
FreeAmp	753K(1.1%)	4K(3.5%)	18K[a]	1[a]	19K[a]	4K(3.5%)	791K(1.2%)	4K(3.5%)
Sensor node by [17]	8K[a]	18[a]	9K[a]	9[a]	1.6M(18.6%)	26(3.7%)	1.6M(18.8%)	103(5%)
MultihopOscilloscope	9K[a]	21[a]	554[a]	9[a]	1.4M(20.8%)	6(3%)	1.4M(20.9%)	110(4%)
CntToLedRfm	8K[a]	18(1%)	3K[a]	6[a]	5K[a]	7(2.9%)	17K[a]	75(4.2%)
BlinkFail	177[a]	7(1.5%)	169[a]	1[a]	1K(4.2%)	6(4.1%)	1K(5.3%)	27(5.8%)
RfmToLed	8K(1.2%)	22(1.2%)	7K(1%)	5[a]	10K(1.5%)	8(2.8%)	26K(3.8%)	76(4.3%)
Packed Malware	1M[a]	7K(3.1%)	5K[a]	2[a]	5K[a]	7K(3.1%)	1M[a]	7K(3.1%)
Multi-Packed Malware	2M[a]	5K(2.2%)	6K[a]	2[a]	7K[a]	5K(2.2%)	2M[a]	5K(2.2%)
BHO	15K[a]	1K(4.7%)	59[a]	4[a]	45[a]	1K(4.7%)	15K[a]	1K(4.7%)

a. Its percentage is less than 1%.

TABLE IV. INSTRUCTIONS CAUSING LOGGING.

virtualization (and hence not present in the original programs). These instructions can be pruned by the symbolic information mapping mentioned in Section IV. Memtrace is a tool that traces all memory addresses. As the addresses accessed in trace program execution are the global memory regions generated by virtualization, PIEtrace has a script that translates such addresses back to the original addresses, leveraging the address mapping *VA*.

We applied two Valgrind tools: Memcheck+ and Callgrind. Memcheck is a tool that detects memory safety problems such as buffer overflows and null pointer accesses. We modified the Memcheck tool to print recent memory read operations as well as instructions that defined the values that are read. We call this new tool Memcheck+ as it is a simple extension of the Memcheck code base. Callgrind is a tool to generate dynamic call graph.

We applied two DynamoRIO tools. Memtrace is similar to Pin-Memtrace. Instcalls is a tool that logs function invocations and returns.

Besides the above tools, we have also applied a number of debugger plugins that provide advanced debugging and profiling capabilities. They are very similar to instrumentation tools. Instead of using instrumentation, they use breakpoints as the mechanism to monitor and inspect program state. For instance, HitTrace logs program state automatically at given breakpoints. Reverse debugging is an advanced feature in x86 gdb that allows reverse execution (e.g. step backward and reverse-continue).

All these tools run correctly with our trace program executions. Next we show a few case studies.

Acrobat Reader 9.3 Acrobat Reader 9.3 on Windows can be crashed by a null pointer dereference when provided with a crafted pdf file. There is not any published explanation of the crash. In this case study, we want to use our simple extension of Linux Valgrind-Memcheck, called Memcheck+, to understand the causality of the crash. We generate the trace program from the crashing execution. We then use Memcheck+ to back-track from the null pointer step by step. We identify the definition point of the value at each step and backtrack to the definition point, till we get to the first such definition. Since Acrobat Reader is highly complex and it does not have any symbolic information, the identified chain is long and crosses a few functions and DLLs, we simplify it in Fig. 4. The boxes on the left show the Memcheck+ output at each step. Each box reports a read instruction including its PC, source location in the trace program, value, and the definition. The corresponding read and write (i.e. the definition) in the original program are shown in the middle. The right column shows the corresponding function and its DLL. At the end, we identify that the definition of the null pointer starts in the *acroform.api* DLL. As shown on the top, the loading of this DLL is guarded by a predicate with branch target 0x4E6180A. Its branch outcome is determined by the input on the right, which is a line in the crafted pdf. We confirm the finding by the fact that any changes to this line makes the crash disappear. Searching it on the Internet, the keywords in the line seems to indicate it is an invalid acrobat form.

Sensor Case. In [17], Qijun Gu et al. show that malicious packets can compromise sensor nodes. We use their program[3] that has a buffer overflow vulnerability and can propagate the malicious packet to other nodes automatically. Using the provided input, the sensor program crashes on an invalid return address. We use PIEtrace to generate the trace program of the crashing run. And then we run the trace program inside a Pin-based dynamic slicer [21]. Specifically, we compute the backward slice from the faulty return address, which contains all the executed instructions that have directly or indirectly contributed to the given faulty value through data and control dependences. The slice has 227 instructions. The data slice (i.e. a slice computed by considering data dependences only) has 38 instructions. The data slice is sufficient to explain the crash. Part of it is shown in the left-hand side of Fig. 5. The corresponding source code is shown on the right. We find that the invalid return address is dependent on `Receive.receive()` (on the bottom), `stackCreator()`, and `strcpy()` (i.e., crashed when returning from the function). This clearly identifies the causal path of the exploit. The malicious packet is received by `Receive.receive()`. It is passed to `stackCreator()`, which calls `strcpy()`. The buffer overflow occurs inside `strcpy()`, corrupting the return address. Note that in a trace program, the original packets become concretized values that are loaded from a file. Hence, the bottom box of Fig. 5 actually corresponds to several IN instructions in TinyOS kernel.

Packed Malware Cases. PIEtrace has the following two main advantages in malware analysis. First, since trace programs do not have any system calls, one can analyze their execution without any concerns about harmful side-effects. Second, since

[3]Downloaded at http://cs.txstate.edu/ qg11/downloads.html

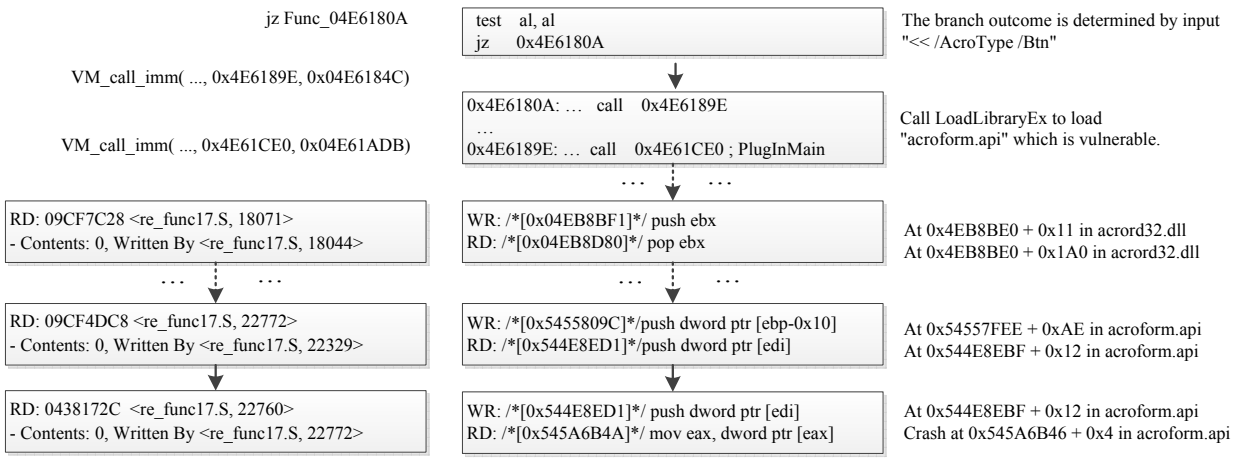

Fig. 4. Analysis results from the Acrobat Reader case study

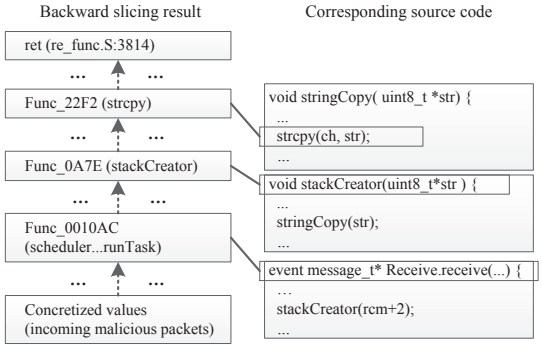

Fig. 5. Backward program slicing on a sensor node.

the technique records all executed instructions, it can be naturally used to analyze packed malwares, which unpack themselves during execution. Compared to existing unpacking techniques [32], [28], [20], [7] that rely on specific heuristics or target specific packers, our technique is very general and easily applicable.

In this experiment, we use a Java runtime installer malware[4] that downloads files to the victim machine to demonstrate this feature. It connects to a server to download another malware. The malware is packed by an unknown version of the UPX [27] executable compressor.

We run the packed malware twice on PIEtrace, one with Internet connection and the other not. In the second execution, the malicious payload is not executed. We get two trace programs. Then, we use the WinMerge tool [26] to compare the two trace programs. The result shows that the first approximately 10,000 basic blocks of the two programs are identical, which implies they belong to the packer and the malware initialization. The differences (about 40,000 basic blocks) denote the malicious payload. From the trace programs, we can also easily observe that the first 678 basic blocks belong to a large loop. Cross checking with the original malware indicates that they belong to the packer and the original malware entry is at the 679th basic block. We can directly execute the malware by putting a goto statement that jumps to the entry. Note that, with our technique, we can easily identify about 9,000 basic blocks that are in the malware but do not perform malicious actions, which cannot be achieved by existing approaches.

In addition, most universal unpackers [32], [20], [7] are not able to handle malware binaries packed multiple times. We call them the multi-packed malware. PIEtrace can directly handle such malware. Hence, in this experiment, we use XPack [19] to further pack the previous packed sample. During execution, the unpacking routines of XPack and UPX are executed sequentially. After unpacking, the actual malware routines will be executed. We repeat the previous process. By comparing the two trace programs, we find that they share about 15,000 basic blocks. The differences (the identified malicious payload) are the same as the previous case. The malware entry point can be easily spotted at the 6781st basic blocks at the end of two large loops.

C. Performance and Scalability

We use SPECINT 2000 to evaluate the performance and scalability of PIEtrace[5]. We could not use the programs in Table II because they are mostly interactive. We run the SPEC programs on test inputs and measure the slow-down of the virtualization component. The results are shown in Fig. 6. For most tests, our system incurs approximately 2000x slow-down or less except `parser` and `crafty`. Further inspection shows that their executions constantly perform I/O, causing a lot of data logging. The average overhead is 2523x. We also evaluate scalability by showing the changes of trace program size and log size over a duration of execution (1,000 million instructions). The results are shown in Fig. 7 and 8. We can observe that the trace program size quickly reaches a fixed point whereas the log size slowly grows over time except `vortex`. Further inspection shows that `vortex` performs more I/O than others in that duration. While we believe we can reduce the runtime overhead by optimization, PIEtrace is a heavyweight technique that may not be used in a production setting. It is suitable for cases where the runtime overhead is less a concern (e.g. in-house debugging), but rather the missing capabilites on the current platform. Such capabilities could be provided by cross-platform tools.

VI. Related Work

Logging and Replay. Logging and replay has been widely studied [2], [37], [36], [42], [6], [29], [43], [40], [12], [15],

[4]Md5 is 6b0bc67c6f87d5e3637fb0e9e0558974 and available at [23]

[5]`Gap` failed to compile in Visual Studio due to library version issues and hence was excluded.

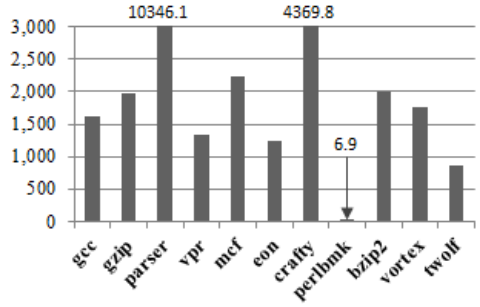

Fig. 6. Slowdown on SPEC INT2000

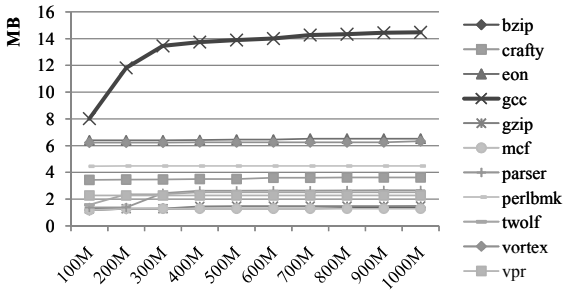

Fig. 7. Trace program size variation.

[33]. However, most of these existing techniques do not support cross-platform replay. Most of them work by intercepting and logging system calls. Replay is by executing *the same program*, with the support of a replay runtime that again intercepts system calls and then loads values from the log. The replayed execution requires the same set of libraries and the same platform. In contrast, we generate a trace program that can be compiled and executed on its own, without any specific platform or runtime support. TRANSPLAY [38] allows replaying a trace on different platforms. However, execution can only be replayed inside a replayer. Hence, debugging a replayed execution discloses the state of the replayer, not the original execution. A replayer has to be developed for each platform. It has to map system calls and library calls across platforms. Creating such mappings requires a lot of manual efforts. In contrast, PIEtrace suppresses platform dependences by concretization. It does not need to understand either the original system interface or the target system interface. S2E [11] allows cross-platform driver execution. Similar to TRANSPLAY, it requires a mapping between kernel APIs across platforms. Xu et al. [41] proposed to use compiler to generate two instrumented versions of a program: one for logging and the other for replay. It works on Java programs and assumes the same set of libraries. In contrast, PIEtrace does not require compilers, and directly works on binaries, which entails a different set of challenges.

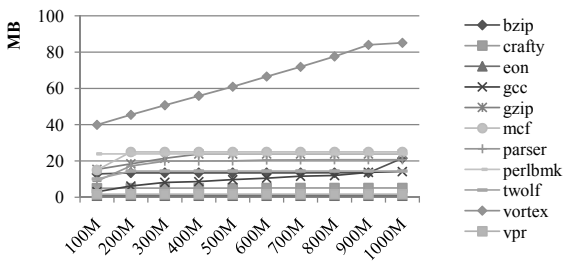

Fig. 8. Log size variation

Tracing. Traditional tracing techniques generate traces [24], [46], [3], [8] that can be analyzed in a cross-platform fashion. However, analysis tools have to support the specific trace formats, which precludes most third party tools. Storing traces is also very expensive.

Virtualization and VM Introspection. There are many virtual machines [4], [13] and emulation infrastructures [31], [1], [10] that allow cross-platform execution. Their main disadvantage is that a subject program has to execute along with their system. It is hence difficult to apply any third party tools to the execution of the subject program. Virtual machine introspection [5], [18], [34] is a kind of technique that aims to observe guest OS state from the host OS. Virtuoso [14] and [16] virtualize utility commands, e.g. `ls`, in the guest OS and make them executable on the host OS so that one can directly observe the state of the guest OS by running such commands on the host OS. These techniques need to be aware of the system interfaces of both OS's and they do not support cross-platform replay of application programs.

Binary Extraction and Reuse. Inspector gadget [22] is a technique that uses dynamic slicing to extract a part of a malware, called a gadget, which can be replayed for behavior analysis. BCR [9] tries to extract components of an executable, such as the decryption/encryption function of a malware, that can be reused in other programs. The extracted gadgets and components are essentially sub-programs that can take inputs and perform certain functionalities such as downloading files. They are platform dependent. Gadget execution also requires support from specific runtime. TOP [45] is a framework that decompiles a binary to C code by executing it. The generated C code can take different inputs as the original binary, whereas PIEtrace only focuses on reproducing a specific execution. TOP is not platform independent since it requires the presence of the same set of libraries, kernel interfaces, external resources and devices.

VII. Conclusion

We propose a novel technique called platform independent executable trace. It generates a standalone trace program from a normal program execution that relies on specific operating system, libraries, hardware, and instruction set. The trace program is platform independent, without relying on any operating system or libraries. It can be compiled and executed on any x86 platform. Running the trace program generates the original execution. As such, the large body of existing third party tools can be applied to analyze trace program execution on the platforms those tools prefer. We have implemented the technique on x86 and sensor platforms. We show that leveraging our technique, Linux tools can be used to analyze Windows and sensor program executions including packed malware executions.

Acknowledgement

This research has been supported in part by DARPA under contract 12011593 and by NSF under awards 0917007 and 1320326. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflct the views of DARPA and NSF.

REFERENCES

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[2] Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In *SOSP '09*, 2009.

[3] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: first fault diagnosis by reconstruction of distributed control flow. In *PLDI '05*, 2005.

[4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05*, 2005.

[5] Chris Benninger, Stephen W. Neville, Yagiz Onat Yazir, Chris Matthews, and Yvonne Coady. Maitland: Lighter-weight vm introspection to support cyber-security in the cloud. In *CLOUD '12*, 2012.

[6] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06*, 2006.

[7] Lutz Boehne. Pandora's bochs: Automated malware unpacking. Master's thesis, RWTH Aachen University, January 2008.

[8] Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *OOPSLA '07*, 2007.

[9] Juan Caballero, Noah M. Johnson, Stephen Mccamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *ISOC NDSS10*, 2010.

[10] Anton Chernoff and Ray Hookway. Digital fx!32 running 32-bit x86 applications on alpha nt. In *NT'97*, 1997.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: a platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 46(3):265–278, March 2011.

[12] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *ATC '08*, pages 1–14, 2008.

[13] Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, 22(1):128–137, May 1994.

[14] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *SP '11*, 2011.

[15] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[16] Yangchun Fu and Zhiqiang Lin. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *SP '12*, 2012.

[17] Qijun Gu and Rizwan Noorani. Towards self-propagate mal-packets in sensor networks. In *WiSec '08*, 2008.

[18] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07*, 2007.

[19] Joko. Xcomp/xpack download page. http://www.soft-lab.de/JoKo/.

[20] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07*, 2007.

[21] Dohyeong Kim. Dualslicing. http://www.cs.purdue.edu/homes/kim1051/.

[22] Clemens Kolbitsch, Thorsten Holz, Christopher Kruegel, and Engin Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *SP '10*, 2010.

[23] Yonghwi Kwon. Platform independent trace project website. https://sites.google.com/site/pitprj12/, May 2013.

[24] James R. Larus. Whole program paths. In *PLDI '99*, 1999.

[25] Peng Li and John Regehr. T-check: bug finding for sensor networks. In *IPSN '10*, 2010.

[26] Christian List, Dean Grimm, Gal Hammer, Jochen Tucht, Kimmo Varis, Alexander Skinner, Takashi Sawanaka, Tim Gerundt, Marcel Gosselin, and Denis Bradford. Winmerge. http://winmerge.org/.

[27] Oberhumer Markus Franz Xaver Johannes, Molnr Lszl, and Reiser John F. Upx: the ultimate packer for executables. http://upx.sourceforge.net/.

[28] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *ACSAC '07*, 2007.

[29] Satish Narayanasamy, Gilles Pokam, and Brad Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *ISCA '05*, 2005.

[30] Joe Polastre. [tinyos-contrib-commits] cvs: tinyos-1.x/contrib/ucb/tos/cc1000pulse cc1000radiointm.nc, 1.2, 1.3. http://mail.millennium.berkeley.edu/pipermail/tinyos-contrib-commits/2005-May/00

[31] Jonathan Polley, Dionysys Blazakis, Jonathan Mcgee, Dan Rusk, and John S. Baras. Atemu: A fine-grained sensor network simulator. In *IEEE SECON '04*, 2004.

[32] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *ACSAC '06*, 2006.

[33] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05*, 2005.

[34] Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In *ICISS '11*, 2011.

[35] Offensive Security. Exploits database by offensive security. http://www.exploit-db.com/.

[36] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS '09*, 2009.

[37] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC '04*, 2004.

[38] Dinesh Subhraveti and Jason Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS '11*.

[39] Berkeley WEBS: Wireless Embedded Systems. download. http://webs.cs.berkeley.edu/tos/download.html.

[40] Amit Vasudevan, Ning Qu, and Adrian Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. In *HICSS '11*, 2011.

[41] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *ESEC-FSE*, 2007.

[42] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA '03*, 2003.

[43] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *MoBS '07*, 2007.

[44] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07*, 2007.

[45] Junyuan Zeng, Yangchun Fu, Kenneth Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *CCS '13*, 2013.

[46] Xiangyu Zhang and Rajiv Gupta. Whole execution traces. In *MICRO '04*, 2004.