PEMU: A PIN Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework

Junyuan Zeng
Department of Computer Science
The University of Texas at Dallas
jzeng@utdallas.edu

Yangchun Fu
Department of Computer Science
The University of Texas at Dallas
yangchun.fu@utdallas.edu

Zhiqiang Lin
Department of Computer Science
The University of Texas at Dallas
zhiqiang.lin@utdallas.edu

Abstract

Over the past 20 years, we have witnessed a widespread adoption of dynamic binary instrumentation (DBI) for numerous program analyses and security applications including program debugging, profiling, reverse engineering, and malware analysis. To date, there are many DBI platforms, and the most popular one is PIN, which provides various instrumentation APIs for process instrumentation. However, PIN does not support the instrumentation of OS kernels. In addition, the execution of the instrumentation and analysis routine is always inside the virtual machine (VM). Consequently, it cannot support any out-of-VM introspection that requires strong isolation. Therefore, this paper presents PEMU, a new open source DBI framework that is compatible with PIN-APIs, but supports out-of-VM introspection for both user level processes and OS kernels. Unlike in-VM instrumentation in which there is no semantic gap, for out-of-VM introspection we have to bridge the semantic gap and provide abstractions (i.e., APIs) for programmers. One important feature of PEMU is its API compatibility with PIN. As such, many PIN plugins are able to execute atop PEMU without any source code modification. We have implemented PEMU, and our experimental results with the SPEC 2006 benchmarks show that PEMU introduces reasonable overhead.

Categories and Subject Descriptors D.3.4 [*Software*]: Processors—Code generation; Translator writing systems and compiler generators; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Design, Security

Keywords Dynamic binary instrumentation; Introspection

1. Introduction

Dynamic binary instrumentation (DBI) is an extremely powerful technique for program analysis. At a high level, it dynamically inserts extra analysis code into the running binary program to observe how it behaves. It works similarly to a debugger but the analysis routine is programmed. Therefore, it can be used to automatically inspect the program state at instruction level and build many program analyses, such as performance profiling (e.g., [42, 46]), architecture simulation (e.g., [29]), program debugging (e.g., [25]), program shepherding (e.g., [23]), program optimization (e.g., [2]), dynamic data flow analysis (e.g., taint analysis [32, 36]), reverse engineering (e.g., [24]), and malware analysis (e.g., [12, 49]).

Today, there are many DBI platforms such as PIN [26], VALGRIND [31], DYNAMORIO [2], QEMU [4], and BOCHS [1]. Each platform is built atop its own virtual machine (VM), and has its own pros and cons. For example, process-level DBI such as PIN and VALGRIND provides rich APIs to analyze user level binary code execution, but the analysis code is executed inside the VM (i.e., in-VM) with the same privilege as the instrumented process. Moreover, it does not support any kernel-level code instrumentation. Some platforms only support a limited type of operating system (OS), e.g., VALGRIND only supports Linux binaries but provides no support for Microsoft Windows binaries. Some platforms are designed as a full system emulator (e.g., QEMU), but do not provide any general DBI APIs. As such, *can we build a cross-OS, API-rich, out-of-VM dynamic binary instrumentation framework that supports both user level and kernel level code?*

While there have been attempts to address this problem, they only partially achieved these goals. Specifically, PINOS [8] attempted to create a kernel instrumentation tool atop the XEN [3] hypervisor. However, it only supports inspecting the very low level instruction semantics (such as the executing instruction address), and does not support any high level instrumentation and introspection (e.g., get the running process ID inside the VM). Meanwhile, because of its implementation of stealing memory from the guest OS, it does not offer strong isolation and the analysis routine can be accessed by the instrumented process or kernel. Another attempt is

VEE '15, March 14–15, 2015, Istanbul, Turkey..
Copyright © 2015 ACM 978-1-4503-3450-1/15/03…$15.00.
http://dx.doi.org/10.1145/2731186.2731201

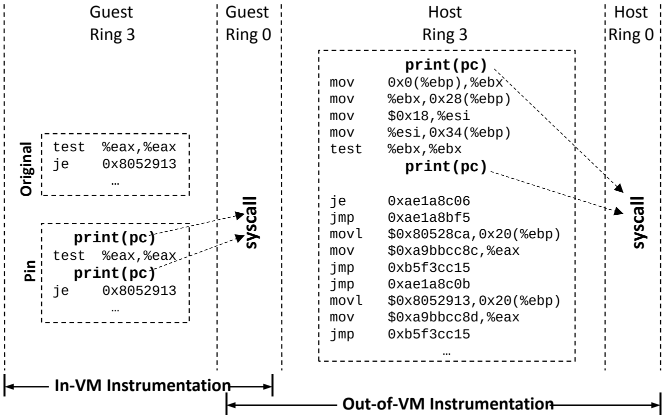
TEMU [48], which extends QEMU with its own APIs to allow end-users to develop TEMU-plugins for whole system instrumentation. Though it has greatly reduced developers' efforts in understanding the internals of QEMU in order to develop any useful plugins, it has only limited APIs compared to those provided by PIN. A recent effort, DRK [34], is able to perform kernel instrumentation. However, it is still an in-VM solution and does not isolate the analysis code (the analysis routine is executed as Linux Kernel modules), resulting in security issues when the kernel has malware.

To address these weaknesses, this paper presents PEMU (inherited from both PIN and QEMU), a new PIN-API compatible DBI framework that provides a whole system instrumentation but from out-of-VM introspection perspective. There are a number of goals PEMU aims to achieve. Specifically, it aims for PIN API-compatibility because of the large amount of users and rich-APIs PIN has. For instance, it has over 450 comprehensive, well-documented, easy to use instrumentation APIs. With the PIN compatible APIs, PIN plugins can be easily ported to PEMU with no or minimal modifications. Meanwhile, it aims for out-of-VM instrumentation because of the isolation requirement from security applications such as introspection. In addition, it aims for supporting a large number of different guest OSes without modification, considering that there are so many OSes with many different versions today.

The key idea to realize PEMU is to add *an additional software layer* atop an existing VM, and make our APIs self-contained. Such a design makes our system easily portable if the underlying VM has been upgraded. In addition to the engineering challenges (in support of the large number of PIN APIs), we also face a number of research challenges. One is how to bridge the semantic gap [9] while providing the out-of-VM instrumentation abstractions (e.g., APIs) for both process and kernel introspection, and also what those abstractions should be. The second is how to design our instrumentation engine such that it works seamlessly with the translation engine provided by the underlying VM but does not introduce large overhead. The third one is how to support the existing PIN APIs by using our framework.

We have addressed these challenges and implemented PEMU atop QEMU, and our experimental results with SPEC 2006 benchmarks show that PEMU has reasonable performance overhead compared to original QEMU and it will be useful for quickly developing PIN-style plugins atop PEMU or directly recompiling the existing PIN-plugins, for both instruction inspection and higher level semantic introspection.

Contributions. In summary, this paper makes the following contributions. We devise an additional software layer atop an existing binary code translation based VM with a set of standard APIs for both user level and kernel level DBI. This additional layer hides the low level VM details and contains a number of instrumentation related abstractions. With the additional layer and the abstractions, we present PEMU, a new DBI framework that enables end-users to develop instrumentation tools using many of the existing PIN APIs. We have implemented PEMU and tested with SPEC 2006 benchmark. Our experimental results show that PEMU introduces reasonable performance overhead.



```
test  %eax,%eax
je    0x8052913
       …
```

```
   print(pc)
test  %eax,%eax
   print(pc)
je    0x8052913
       …
```

```
          print(pc)
mov    0x0(%ebp),%ebx
mov    %ebx,0x28(%ebp)
mov    $0x18,%esi
mov    %esi,0x34(%ebp)
test   %ebx,%ebx
          print(pc)

je     0xae1a8c06
jmp    0xae1a8bf5
movl   $0x80528ca,0x20(%ebp)
mov    $0xa9bbcc8c,%eax
jmp    0xb5f3cc15
jmp    0xae1a8c0b
movl   $0x8052913,0x20(%ebp)
mov    $0xa9bbcc8d,%eax
jmp    0xb5f3cc15
            …
```

Figure 1. Differences Between in-VM and Out-of-VM Instrumentation.

2. Background and Overview

In this section, we first discuss the background related to our system in §2.1, and then motivate our research in §2.2. Next, we discuss how to develop a plugin using PEMU in §2.3, and finally we give an overview of PEMU in §2.4.

2.1 In-VM vs. Out-of-VM Instrumentation

The key technique behind any DBI is the just-in-time compilation (JIT) [4, 37]. Basically, all the executing instructions are translated by a JIT compiler, which provides an opportunity to interpose and instrument the binary code for analysis purposes. The entire DBI infrastructure can be considered as a VM, which could be a process level VM (e.g., PIN, VALGRIND), or a system level VM (e.g., QEMU). At a high level, a VM mediates program execution by dynamically translating blocks of native code and executing them from a code cache.

Given an analysis routine (e.g., printing the executed instruction addresses), there are two different ways of instrumenting the analysis routine with the original program code, as illustrated in Fig. 1.

- **In-VM instrumentation**. This is the easiest way. The analysis routine is directly translated together with the original code into the same code cache. The analysis routine and the original program code share the same address space, and they are executed inside the VM either at guest "ring 3" (application layer) or "ring 0" (OS kernel layer). Therefore, the analysis routine can feel free to call any guest OS abstractions, and access any code or data of the instrumented process or kernel. Most DBI systems (e.g., PIN and VALGRIND) are designed in this way.

```
1  static UINT64 icount;
2  FILE *pFile;
3  VOID docount(UINT32 c) { icount += c; }
4  VOID Trace(TRACE trace, VOID *v) {
5     for (BBL bbl = TRACE_BblHead(trace);
6        BBL_Valid(bbl); bbl = BBL_Next(bbl)) {
7        BBL_InsertCall(bbl, IPOINT_BEFORE,
8        (AFUNPTR)docount, IARG_UINT32, BBL_NumIns(bbl),
9        IARG_END);
10    }
11 }
12 VOID Fini(INT32 code, VOID *v) {
13    fprintf(pFile, "Count %lld\n", icount);
14    fclose(pFile);
15 }
16 INT32 Usage(VOID) {
17    return 0;
18 }
19 int main(int argc, char * argv[]) {
20    if(PIN_Init(argc, argv)) return Usage();
21    pFile = fopen("pemu_count", "w");
22    TRACE_AddInstrumentFunction(Trace, 0);
23    PIN_AddFiniFunction(Fini, 0);
24    PIN_StartProgram();
25    return 0;
26 }
```

Figure 2. A PEMU plugin to count the number of executed instructions.

- **Out-of-VM instrumentation**. Unlike in-VM instrumentation, the analysis routine is executed outside of the original program code (mostly at the virtual machine monitor layer, e.g., at "ring 3" of a host OS), though the original code and the analysis routine can be translated into the same cache. Therefore, the analysis routine and the original code does not share the same address space any more. There is a world switch for analysis routine from host "ring 3" to access the state of the monitored process or kernel at "ring 3" or "ring 0" of guest OS. Only a handful of systems (e.g., PINOS and TEMU) support out-of-VM instrumentation. However, their introspection supports are guest OS specific. Also, the isolation provided by PINOS is not as strong as TEMU. More specifically, while the instrumented code and instrumentation engine do not share any code in PINOS, the analysis routine and the original program code actually share the same address apace, because the analysis routine steals [8] the address space from the guest OS, which makes it possible to tamper with the analysis routine when used to analyze malware.

We can notice that in-VM instrumentation and out-of-VM instrumentation share the opposite pros and cons. In-VM instrumentation occurs inside the VM, and has rich abstractions. But it executes at the same privilege level as the monitored process. Out-of-VM instrumentation occurs outside of the VM, and has less abstractions. But the analysis routine is isolated with the original program code. To develop the analysis routine, we can still use host OS abstractions, but to inspect any guest OS state, there is a need for techniques to bridge the semantic gap.

2.2 Objectives

While there have been significant efforts in the past 20 years to build various DBI platforms, few works focus on out-of-VM instrumentation where the analysis routine and original program code are strongly isolated. In this paper, we would like to develop a new out-of-VM DBI with an emphasis on supporting security applications that satisfy the following constraints:

(1) **Rich APIs**. Similar to PIN tool, we would like to offer rich and well-defined APIs. Since PIN is one of the most popular DBI tools, we would like to make our API compatible with PIN. This will make an open source alternative to PIN, which will be useful when there is a need to customize the PIN engine.

(2) **Cross-OS**. Unlike VALGRIND, which only analyzes Linux binaries, we would also like to offer support to instrument both Windows and Linux binaries using the same platform. More importantly, since there are a large number of different OSes, we would like to make our system OS agnostic for the introspection.

(3) **Strong Isolation**. Unlike existing in-VM approaches, we would like to make our analysis code execute at the hypervisor layer (can be considered "ring -1") instead of at the guest OS "ring 3" for process introspection or "ring 0" for kernel introspection.

(4) **VM Introspection.** Unlike PINOS, which does not support higher level guest object introspection [16], we would like to provide APIs to retrieve the high level semantic state from the guest OS for the monitored process or guest kernel. Considering that there are too many guest OSes, we would like to design a general way to query the guest OS state.

2.3 An Example

Before describing the details of how we achieve these goals, we would like to first illustrate how to develop a PEMU plugin by using the provided APIs. As presented in Fig. 2, this is a very simple plugin with the functionality of counting the number of executed instructions. Similar to many other DBIs, to develop a PEMU plugin, users need to provide two sets of procedures: *Instrumentation Routine*, which specifies where the instrumentation should occur; and *Analysis Routine*, which defines analysis activities.

One important feature of PEMU is the API compatibility with the PIN tool. As illustrated in this example, the API we used is exactly identical to those used by a PIN tool. Therefore, many legacy PIN plugins can be recompiled and executed inside PEMU, but the distinctive feature is that both the analysis routine and instrumentation routine will be executed in the host OS, instead of inside the guest OS as would be done using PIN. For instance, the `fprintf` (line 13) and `fopen` (line 21) statements will be executed at

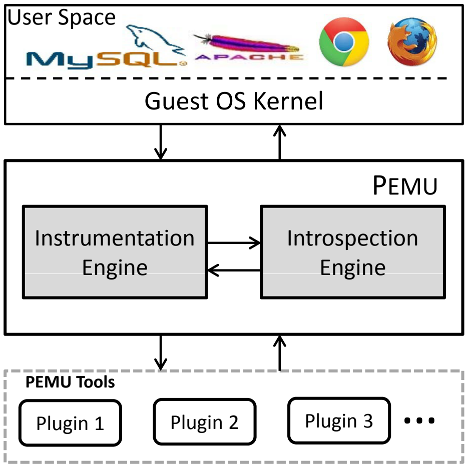


Figure 3. Architecture Overview of PEMU.

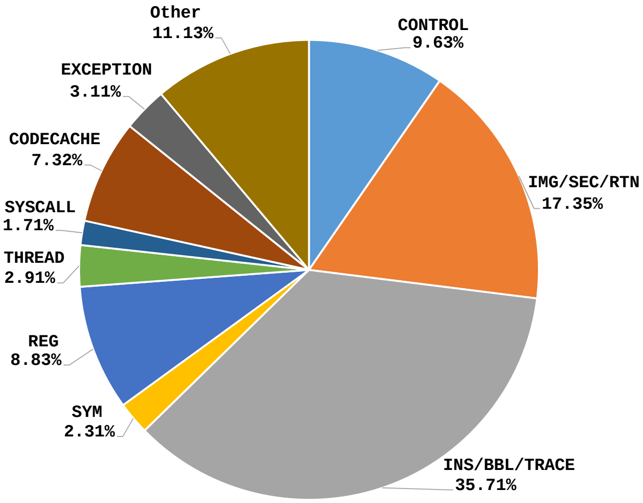


Figure 4. Distributions of PIN APIs.

the VMM layer and call host OS `fprintf` and `fopen`, but when we use PIN they will be executed inside the guest OS.

2.4 Architecture Overview

An overview of PEMU is presented in Fig. 3. There are two key components inside PEMU: *instrumentation engine* (§3) and *introspection engine* (§4). They both are located inside a virtual machine monitor.

To use PEMU, end users use the PIN compatible APIs provided by our *instrumentation engine* to develop the plugins, which will be compiled and linked at the VMM layer (namely, the host OS layer). If the analysis routine requires retrieving the state of the guest OS (e.g., `pid` of the instrumented process), it uses APIs provided by our *introspection engine*. In the following two sections, we describe how we design these two engines in greater detail.

3. Instrumentation Engine

Since PEMU aims for API compatibility with PIN, we have to first examine what those PIN APIs are. We take a recently released version of PIN (version 2.13), and we find there are in total 477 APIs. The distribution of these APIs are presented in Fig. 4.

PIN defines two sets of instrumentation: (1) *trace instrumentation* that occurs immediately before a code sequence is executed, and (2) *ahead-of-time instrumentation* that caches the instrumentation before the execution. There are three different types of *trace instrumentation* depending on the granularity:

- **Instruction Level**. The finest granularity is the instruction (INS) level instrumentation that allows for instrumenting a single instruction at a time by using the `INS_AddInstrument Function` call back. There are also many instruction insertion and inspection APIs starting with the `INS` prefix (e.g., `INS_InsertIfCall`, `INS_IsBranch`, etc.). In total there are 142 `INS` related APIs.
- **Basic Block Level (BBL)**. A basic block (BB) is a single entrance, single exit sequence of instructions. Instead of one analysis call for every instruction, it is often more efficient to insert a single analysis call for a BB, thereby reducing the number of analysis calls. PIN does not offer a `BBL_AddInstrumentationFunction`, and instead developers have to instrument the TRACES (described next) and iterate through them to get the BB. There are in total 14 APIs related to BBL.
- **Trace Level**. A TRACE in PIN is defined as a sequence of instructions that begin at the target of a taken branch and end with an unconditional branch (i.e., `jmp/call/ret`). This is the set of instructions that are disassembled by a linear sweep algorithm, when giving a starting address. Therefore, a TRACE usually consists of a number of BBs. PIN provides `TRACE_AddInstrumentFunction` call back to instrument a TRACE. There are in total also 14 APIs related to TRACE. Note that PIN introduced the concept of TRACE for a trace-linking optimization [26], which attempts to branch directly from a trace exit to the target trace without trapping to the VM. TRACE is at a higher granularity than BB and INS, and sometimes instrumenting analysis routine at TRACE granularity can improve performance. For instance, TRACE-based BBL instruction counting (as shown in Fig. 2) is much faster than that of an INS based one.

Regarding the *ahead-of-time instrumentation*, PIN provides an image (IMG) instrumentation and routine (RTN) instrumentation. More specifically:

- **IMG instrumentation** allows a PIN-tool to inspect and instrument an entire image when it is loaded. A PIN-tool can walk the sections (SEC) of an image, the RTN of a section, and the INS of a routine. Image instrumentation relies on symbol information to determine an RTN boundary. An analysis routine can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. IMG instrumentation utilizes the `IMG_AddInstrumentFunction` API. In total, there are 27 APIs related to IMG, and 16 APIs related to SEC.

- **RTN instrumentation** allows a PIN-tool to inspect and instrument an entire routine when the image, it is contained in, is first loaded. A PIN-tool can walk the instructions of an RTN. An analysis routine can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. RTN instrumentation utilizes the `RTN_AddInstrumentFunction` API. In total, there are 39 APIs related to RTN.

Next, we discuss how to design PEMU in support of these APIs. As we base PEMU atop QEMU, we have to examine the difference between QEMU and PIN. In fact, there are substantial differences, leading to a number of new challenges while designing PEMU.

First, QEMU does not introduce any abstractions for TRACE, SEC, RTN, and IMG, and it only allows instrumentation at the INS or BB level. Therefore, we have to rebuild these missing abstractions. Second, QEMU's disassembling is based on BB, and the size of a BB has a limited value. For example, we notice that in QEMU-1.53 a BB needs to be split if the number of generated intermediate instructions is greater than 640. But there is no such constraint in PIN.

To address these challenges, we add our own disassembler rather than using the one in QEMU. Our own disassembler aims to reconstruct the abstractions for TRACE and we thus call this component *TRACE Constructor* (§3.1). To insert the analysis routine into the original program code, we leverage QEMU's dynamic binary translation (DBT) engine, on top of which to design our *Code Injector* (§3.2). In the rest of this section, we present the detailed design of these two components.

3.1 TRACE Constructor

The fundamental reason to introduce our own disassembler is to build the TRACE abstraction for PIN-APIs, from which to further build many other APIs such as those related to RTN, BB, and INS. Meanwhile, to uniformly support both trace and ahead of time instrumentation, we use a cache (we call hooking point hash table) to store all the call-back points where an analysis routine is instrumented. Then whenever these instruction points are executed, they will automatically invoke the analysis routine defined by users.

Since we aim to build TRACE abstractions, which contain BB and INS, we have to disassemble per TRACE. However, QEMU disassembles an instruction at a time (per BB). Therefore, when a starting address of a TRACE is to be disassembled by QEMU, we will disassemble all of the following instructions until we encounter an unconditional branch (e.g., `jmp`, `ret`, `call`), which is the end of a TRACE. As such, we will hold an entire TRACE before QEMU disassembles each instruction inside it.

```
1:  Global: $TPC$: a set storing the starting address of a $TRACE$;
    $HPHT$: the global hooking point hash table indexed by $PC$ and
    storing function pointers of the user defined analysis routine.
2:  Input: $PC$, the current trace starting instruction address;
3:  Output: a $TRACE$, and updated $TPC$ and $HPHT$.
4:  Disassemble ($PC$) {
5:      $TRACE \leftarrow$ GetTRACE();
6:      $BB \leftarrow$ GetBB();
7:      do {
8:          $INST \leftarrow$ DisasINST($PC$);
9:          InsertINST($BB$, $INST$);
10:         if (INST_Instrument $\neq$ NULL) {
11:             INST_Instrument($INST$, $HPHT$);
12:         }
13:         if ($INST$.type $\in$ {jcc, jmp, call, ret}) {
14:             InsertBB($TRACE$, $BB$);
15:             if (BB_Instrument $\neq$ NULL) {
16:                 BB_Instrument($BB$, $HPHT$);
17:             }
18:             $BB \leftarrow$ GetBB();
19:             $TPC \leftarrow TPC \cup$ GetTargetPC(INST) ;
20:         }
21:         $PC \leftarrow PC + INST$.InstLen();
22:     } while($INST$.type $\notin$ {jmp, call, ret});
23:     if (TRACE_Instrument $\neq$ NULL) {
24:         TRACE_Instrument($TRACE$, $HPHT$);
25:     }
26: }
```

Algorithm 1: *TRACE Construction*

However, there are also some practical challenges. One is that the instructions that belong to a TRACE may not exist in the guest OS memory (swapped or not loaded yet). The other is the instructions being disassembled are not currently being translated by QEMU-DBT, which is the underlying component for our Code Injector. Consequently, we cannot insert our analysis routines into the guest code while perform our disassembling.

To solve the first challenge, we use a proactive page fault injection approach that is triggered from the hypervisor layer and let the guest OS map the missing pages. For the second challenge, we use a global hooking point hash-table (HPHT) to cache the instruction point that will have analysis routine inserted. Later, when QEMU generates the translated code, our *Code Injector* will query this hash table to insert the analysis routine if there is any.

The Algorithm. To precisely describe how we build the TRACE abstraction and facilitate the instrumentation process, we use Algorithm 1 to show its details. For each PC that is a trace starting address, we will start disassembling the whole TRACE (line 4-26). This is the only point to invoke our own disassembler. To decide whether a given PC is a trace starting address, we query the TPC set that stores all the starting addresses of the TRACES. Note that some of the starting

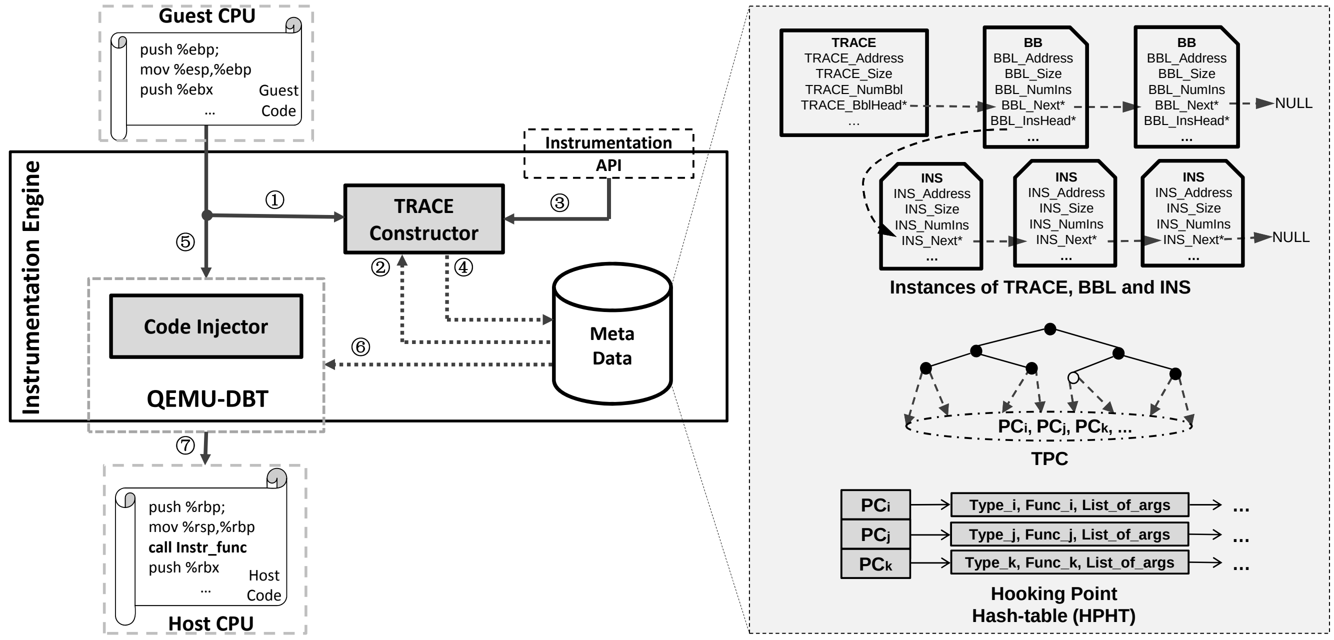


Figure 5. Detailed Design of Our Instrumentation Engine.

address is dynamically computed, especially for indirection control flow transfers.

To disassemble a TRACE, we first create a TRACE (line 5) and a BB instance (line 6), respectively. Then we disassemble and iterate each instruction inside the basic block, and add them into the corresponding BB (line 8-9). If there is any instruction level instrumentation (e.g., when `INS_InsertCall` is called in the PEMU plugin), we add the hooking point of the disassembling instruction into the HPHT (line 10-12). Next, if there is a control flow transfer instruction (line 13-20), then the current BB ends and we insert it into the current TRACE (line 14). Also, we insert a BB hooking point if there is any (line 15-17). Meanwhile, we allocate a new BB (line 18). To get a new TRACE starting address, we invoke a helper function, GetTargetPC (line 19), and we store the new starting address in our TPC. Next, we continue to get the next instruction (line 21), which can be the next instruction inside a BB or a starting address of a new BB. Until we encounter an unconditional control flow transfer instruction, we finish disassembling the current TRACE. If there is any TRACE instrumentation, we add the TRACE hooking points into HPHT (line 23-25).

Regarding the connection between TRACE, BB, and INS, we illustrate their data structures in the right hand side of Fig. 5. Each instance of these data structures is semantically compatible with the corresponding PIN counter-part. With these data structures, PIN's instrumentation and inspection APIs can be easily implemented. For example, when `BBL_Next` is called in a plugin, we will correspondingly traverse the BB instance list to return the next BB.

3.2 Code Injector

To inject the analysis routine that is specified in our HPHT, we leverage the QEMU's DBT for this goal. In particular, to translate the guest binary code into host code, QEMU uses a tiny code generator (TCG), which provides APIs to insert additional code. Having collected which instruction needs the instrumentation, our *Code Injector* will directly use the TCG API (e.g., `tcg_gen_helper`) to insert the analysis routine.

We can also notice that reconstructing TRACE abstractions as well as using the HPHT significantly alleviates the complexity of the instrumentation. With these data structures, we can uniformly achieve code injection anywhere regardless of the granularity. For instance, we can inject an analysis routine at an entry address of a BB, starting address of a TRACE, entry or exit address of an RTN, or just a particular instruction address. That is why we do not attempt to construct abstractions for RTN, SEC, and IMG. For them, we just perform ahead-of-time disassembling and extract the instruction address of interest. For instance, to hook the `malloc` routine, we just need to know the entry address of this function (which can be acquired by signature scanning in the guest memory), and then at runtime, we inject the analysis routine if there is a need for the function argument interpretation of `malloc`.

3.3 Putting it all together

To put it all together, we illustrate the overall execution steps of our *instrumentation engine* in Fig. 5. For each guest instruction, our *TRACE Constructor* will take control (**Step** ①). It first checks whether the current instruction is a starting address of a TRACE by querying the metadata (**Step** ②) that stores all the observed tracing starting addresses. Note that to disassemble a new TRACE, its starting address must have been observed by QEMU, and therefore it has already been included in our TPC set (we use a red-black tree to store this set). If this is not a trace starting address, then we directly continue the execution of *Code Injector* (**Step** ⑤) to generate the final host code (**Step** ⑦). During the code generation, our *Injector* may query the metadata, especially the HPHT data structure, to decide whether the current instruction needs an instrumentation (**Step** ⑥).

If the instruction is a TRACE starting address, then our disassembler will be invoked to disassemble the entire TRACE. During the disassembling, it will insert the corresponding instrumentation routine into the entry of the HPHT (**Step** ④), if such a routine is specified by instrumentation API in the user defined plugins (**Step** ③). When the disassembling finishes, the execution continues to Code Injector (**Step** ⑤) to generate the final host code.

4. Introspection Engine

Since the plugin of PEMU is executed below the guest OS, we have to design an introspection engine that supports the identification of the monitoring process/threads, as well as bridges the semantic gap when the plugins need to inspect the state of the monitored process or OS kernels.

4.1 Identification of Monitored Process/Threads

The instrumentation APIs and the execution of the analysis routine need to be executed when the monitored process is executing. In PIN, all of them are executed in the same address space as the monitored process. However, in PEMU, all of them are executed below the guest OS. Therefore, we have to precisely identify the target process or threads.

Given a running OS, there are a number of ways to differentiate and retrieve the process or thread execution context from a hypervisor layer. One intuitive approach is to traverse kernel data structure to locate the process name, but such an approach is OS-gnostic. Other approaches include using the value of page global directory (PGD) to differentiate each process (as shown in [21, 22]), or using PGD and the masked value of the kernel stack pointer (as shown in [15]).

In PEMU, we adopted the PGD and kernel stack pointer approach. However, we still need to extend it to capture the beginning of the process/thread execution because our instrumentation happens right before process execution. To this end, we propose to capture the data life time of PGD to identify the new process. This is based on the observation that the guest OS must allocate a new (unused) PGD when creating a new process. In x86, PGD is stored in control register CR3. Therefore, if we keep tracking the use of CR3, we can detect a new process.

More specifically, starting from the execution of the guest OS, we maintain a list of the used CR3 values. When a new value is used to update the CR3 (by monitoring `mov` instructions with the destination register `cr3`), we detect that a new process is created. However, we also need to capture when the process exits, because a dead process's CR3 value can be reused for new process. Therefore, we also monitor the execution of `exit` syscall, and the CR3 value used in this syscall will be removed from the CR3 list such that we can detect a new process when this value is used again.

Note that all threads share the same address space. Therefore they will have the same CR3. To differentiate threads, we use the masked value of kernel stack pointer, because each thread will have a corresponding kernel stack that keeps the return addresses and local variables of the functions executed in a syscall trapped from the thread.

4.2 Addressing the Semantic Gap Challenge

Once we have detected the newly created process/thread, our instrumentation will be performed on the monitored process/thread if the instrumentation is for it. Nearly all of our instrumentation APIs are self-contained, and many of them use the abstractions provided by the host OS. Therefore, for most of the instrumentation and analysis routines, there is no semantic gap. For instance, the analysis routine can call `fprintf` in the host OS to print the analysis result.

Unfortunately, for analysis routines that inspect the running process or kernel states, we have to reconstruct their abstractions (namely bridging the semantic gap). For instance, we cannot call the `getpid` syscall at the VMM layer, because the return value of this syscall will be the `pid` of the VMM. Instead, we need to retrieve the `pid` of the monitored process executed inside the VMM.

In the past decade, many approaches have been proposed to address the semantic gap challenge. These approaches include leveraging the kernel debugging information, as shown in the pioneer work Livewire [16]; analyzing and customizing kernel source code (e.g., [18, 35]); manually creating the routines to traverse kernel objects based on kernel data structure knowledge (e.g., [20, 33]); or using a dual-VM based binary code reuse approaches [11, 14, 15, 47]. Some of these approaches (e.g., [14, 18, 20]) have a strong semantic gap [19], which does not trust any guest OS code; Some of them (e.g., [47]) have a weak semantic gap, which trusts guest kernel code, but not application code.

To make PEMU more practical, we adopt the approach proposed in HyperShell [47]. Though it is a weak semantic gap approach, it is guest OS agnostic. More specifically, by taking this approach, we will forward the syscall execution into the guest OS if the syscall needs to inspect or retrieve the guest OS state. Regarding which syscall needs such forwarding, we let the PEMU plugin developers decide but we provide the corresponding APIs for them. For instance, if a plugin needs to retrieve the instrumented process ID, the plugin developers will invoke `PEMU_getpid`. If a plugin needs to open a file in the guest OS, it will use `PEMU_open`, and this file will be closed by `PEMU_close`. In other words, we provide a set of wrapper functions with `PEMU` prefix for state inspection

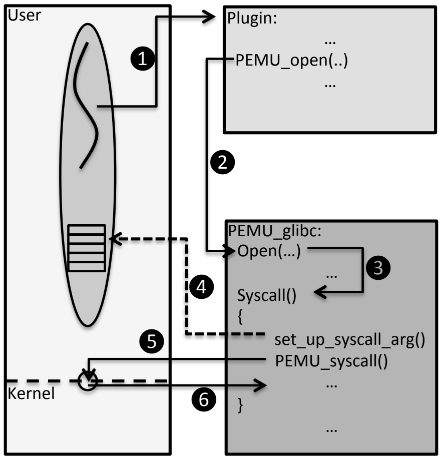


Figure 6. Detailed Steps For An Execution Forwarded Guest Syscall.

and file system related `glibc`-APIs. These APIs work as usual except that we have to detour the control flow of the entry point and exit point of these syscalls, such that the corresponding syscall execution can be forwarded to the guest OS. In total, there are 28 state inspection syscalls (including `getpid,gettimeofday` etc.), and 15 file system related syscalls (including `open,fstat,lseek`,etc.), which are forwarded to the guest OS if the plugin uses the PEMU prefix syscalls.

Though PEMU offers a weak semantic gap, we would like to note that for all other syscalls involved in the analysis routine, we offer a strong semantic gap. This is because we will not execute any guest code, will not traverse any guest kernel data structures, and the execution of the syscall will be directly executed on the host OS. If there is a strong security need, only the results for syscalls prefixed with PEMU cannot be trusted. In other words, a plugin developer is aware of this and can hence quantify the trustworthiness of her analysis routine.

Execution of a Forwarded Guest Syscall. To illustrate how a forwarded syscall really works, we present its detailed execution steps in Fig. 6. In general, there are three parts of code involved in an introspection process: (1) original program code, (2) the analysis routine, and (3) the modified `PEMU_glibc`.

Suppose the control flow is transferred to an analysis routine (**Step ❶**), which needs to open a file inside the guest VMby calling `PEMU_open`. Then, `PEMU_open` goes to the real `open` in `PEMU_glibc` (**Step ❷**). Next, it invokes the `syscall` function (**Step ❸**) where real `sys_open` is triggered. PEMU intercepts `syscall` so that it will not trap to host OS kernel. To forward the syscall execution to guest OS, it first needs to save the register context and set up the arguments (**Step ❹**). If the argument is a pointer, we cannot directly pass that pointer to the guest VM because the guest OS can only access memory in its address space. To allow legal memory access inside the guest OS, we inject a `sys_mmap` to allocate a piece of memory and copy the argument content to the allocated memory (here it is the file name in this case). Next, it waits until the instrumented process executes in user space, and then it forces the execution of the syscall entry (**Step ❺**). The control flow goes back to the original program and a forwarded syscall gets executed. Finally, right after the execution of the syscall exit, PEMU copies the result and restores the register context (**Step ❻**).

5. Evaluation

We have implemented a proof-of-concept prototype of PEMU atop `qemu-1.5.3`. We use `XED` library to build our own disassembler. Meanwhile, we have implemented over one hundred PIN compatible instrumentation APIs for INS, RTN, BB, and TRACE, as well as 43 guest OS state inspection and file system related APIs. To implement the rest of the APIs is an engineering challenge, and we leave it for future work.

In this section, we present our evaluation result. We first test the compatibility of PEMU with PIN in §5.1. Then in §5.2 we evaluate the performance of PEMU using an instruction count plugin (shown in Fig. 2) with the `SPEC CPU2006` benchmark. Next, we evaluate the memory cost of PEMU in §5.3. Finally, we perform case studies to demonstrate the unique benefits of PEMU in §5.4. Our host environment runs `Ubuntu 12.04` with 32-bit Linux kernel `3.0.0-31-generic-pae`, on Intel Core i7 CPU with 8G memory. Our guest OS is a 32-bit `Ubuntu 11.04` (Linux kernel `2.6.38-8-generic`) with 512M memory.

5.1 Compatibility Testing With PIN Plugins

To test how compatible PEMU is with PIN, we download the most recent released PIN tool, and use the plugins in `SimpleExamples` directory for this test. In total, there are 23 plugins. We recompile these plugins with PEMU's header files and library files. As shown in Table 1, a pleasant surprise is that 21 of them can be executed without any problem, considering that so far we only implemented over one hundred PIN APIs.

More specifically, we notice that most of these test plugins are mainly used to test the tracing of instructions (including opcode and operand), control flow transfers (branching, call, ret, etc.), memory access, and library calls. Since these are the basic functionalities for a DBI tool, the current implementation of PEMU fortunately supports all of them.

As shown in Table 1, we have two failures `dcache.so` and `opcodemix.so`. The main reason is that our current implementation does not support APIs for `CODECACHE` and `CONTROLLER`. Note that `CODECACHE` allows developers to inspect the PIN code cache and/or alter the code cache replacement policy, and `CONTROLLER` is used to detect the

Plugin	Description	Supported
calltrace.so	Call trace tracing	✓
extmix.so	Instruction extension mix profile	✓
inscount2_vregs.so	Counting executing instructions	✓
pinatrace.so	Memory address tracing	✓
xed-cache.so	Decode cache profile	✓
catmix.so	Instruction category mix profile	✓
fence.so	Runtime text modification guard	✓
jumpmix.so	Jmp/branch/call profiling	✓
regmix.so	Register usage mix profile	✓
xed-print.so	XED usage testing	✓
coco.so	Code coverage analyzer	✓
icount.so	Counting executing instructions	✓
ldstmix.so	Register/memory operand profiler	✓
topopcode.so	Opcode mix profiler	✓
xed-use.so	XED interface usage testing	✓
dcache.so	Data cache simulation	✗
ilenmix.so	Instruction length mix profiler	✓
malloctrace.so	Tracing calls to malloc	✓
toprtn.so	Hostest routines profiling	✓
edgcnt.so	Control flow edge profiler	✓
inscount2_mt.so	Counting executing instructions	✓
opcodemix.so	Opcode mix profiler	✗
trace.so	Compressed instruction tracer	✓

Table 1. Compatibility Testing with Existing PIN Plugins.

beginning or end of an interval of the execution of a program. We leave the support of these APIs for future work.

5.2 Performance Evaluation

Next, we test the performance of PEMU. We perform two sets of experiments: one is to measure how slow PEMU is when compared to a vanilla-QEMU, and the other is how slow when compared to PIN. We directly use the instruction counting plugin described in Fig 2. This plugin increases the number of instructions in a BB for an accumulated counter before the execution of each BB. We test this plugin with the SPEC 2006 benchmark programs. Each of the benchmark programs is executed 100 times, and we use the corresponding average number in our report.

Performance Comparison with vanilla-QEMU. In this experiment, we measure the overhead introduced by PEMU instrumentation. We compare the execution when running the benchmarks with PEMU, directly with QEMU without any instrumentation.

We report the detailed experimental result in Table 2. Specifically, we show the total number of instructions executed in the 2^{nd} column and also the execution time of QEMU and PEMU is reported in the 3^{rd} and 4^{th} column (namely, T_{Qemu} and T_Pemu). We notice that on average there are 17649.1 million instructions traced for these benchmarks, and the average slowdown over QEMU is about 4.33X, which we believe it is reasonable for practical use. This overhead includes our TRACE Constructor, Code Injector, as well as runtime overhead of the analysis routine.

Performance Comparison with PIN. In the second experiment, we compare PEMU against PIN using the same plugin with the same benchmark. The execution time of running in PIN is presented in the 6^{th} column, and the comparison between PEMU and PIN is presented in the last column.

We notice that the average slowdown between PEMU and PIN is over 83.61X. The main reason is that PIN is running natively while PEMU (based on QEMU) needs extra translation. The largest slowdown comes from `444.namd` which is above 310X. However, we note that when running this program in vanilla-QEMU, it will have close to 100X slowdown. We carefully examine the reason and find the root cause due to the use of large amount of floating point instructions which needs time-consuming emulation inside QEMU.

It is also interesting to note that for `450.soplex`, running in QEMU is faster than that of PIN. The main reason is this program contains more control flow instructions that will go to the middle of a TRACE, thereby breaking the TRACE. In this case, QEMU (based on BBL disassembling) will just redisassemble the basic block that has not been disassembled, but PIN (based on TRACE disassembling) will redisassemble the whole trace after a new trace is found. Meanwhile, the running time of this program is relatively short. Thus, the time is dominated by the disassembling time.

5.3 Memory Cost Evaluation

Since PEMU uses an ahead-of-time instrumentation that will store the hooking point to facilitate the instrumentation, we would like to measure how much memory space this hooking point table consumes. Again, we evaluate this memory cost with our instruction counting plugin against the SPEC2006 benchmark. The result is presented in Fig. 7. We notice that the average memory cost is about 9M.

More specifically, as shown in Fig. 7, the maximum memory cost comes from `465.tonto` (about 22M) because this program contains the largest number of `BB`, resulting in the largest hash table to store the hooking points. More interestingly, `464.h264ref` is one of the most time consuming programs but requires a relative small size of hash table. The reason is that this program contains lots of `loops` and thus certain instructions get executed repeatedly.

5.4 Case Studies

We have demonstrated using PEMU to analyze Linux binaries. In fact, our system is cross-OS, which is one of our design goals. To test this, we apply PEMU to analyze Windows binaries as we have evaluated with Linux binaries. In particular, we use a number of anti-PIN binaries during this test.

First, we test how PEMU would analyze the software protected by `tElock` and `safengine shielden`, which are two widely used tools to build anti-analysis software. We apply these protectors to the `hostname` binaries in a Win-XP SP3 machine, with anti-debugging and anti-instrumentation enabled, and produce two anti-analysis

Program	#Inst (M)	T_{Qemu} (s)	T_{Pemu} (s)	T_{Pemu} / T_{Qemu}	T_{Pin} (s)	T_{Qemu} / T_{Pin}	T_{Pemu} / T_{Pin}
401.bzip2	11500.27	24.55	81.15	3.31	11.17	2.20	7.26
403.gcc	4940.36	18.35	169.21	9.22	13.56	1.35	12.48
410.bwaves	29360.09	419.99	1336.57	3.18	7.44	56.45	179.65
416.gamess	2121.15	23.19	84.99	3.66	3.35	6.92	25.37
429.mcf	3562.67	23.91	70.58	2.95	3.55	6.74	19.88
433.milc	39509.49	779.07	2570.44	3.30	9.28	83.95	276.99
435.gromacs	4907.53	106.28	334.74	3.15	3.43	30.99	97.59
436.cactusADM	9730.11	304.89	1019.89	3.35	4.42	68.98	230.74
437.leslie3d	55857.54	900.01	3009.06	3.34	15.38	58.52	195.65
444.namd	74037.63	1523.78	5037.00	3.31	16.22	93.94	310.54
445.gobmk	314.88	2.43	4.17	1.72	1.71	1.42	2.44
450.soplex	63.67	1.49	2.22	1.49	1.80	0.83	1.23
453.povray	2987.41	36.16	193.17	5.34	3.52	10.27	54.88
454.calculix	187.33	2.53	6.61	2.61	2.46	1.03	2.69
456.hmmer	17862.2	46.43	260.56	5.61	6.95	6.68	37.49
458.sjeng	15514.49	48.40	432.79	8.94	14.38	3.37	30.10
462.libquantum	408.63	0.77	2.01	2.61	0.62	1.24	3.24
464.h264ref	98144.32	392.21	2751.31	7.01	34.01	11.53	80.90
465.tonto	3571.85	48.23	195.16	4.05	5.44	8.87	35.88
470.lbm	7744.81	161.22	692.51	4.30	2.92	55.21	237.16
471.omnetpp	2209.23	16.24	136.63	8.41	3.19	5.09	42.83
473.astar	26645.10	102.95	734.52	7.13	13.67	7.53	53.73
482.sphinx3	6198.21	77.95	322.26	4.13	4.94	15.78	65.23
999.specrand	6198.21	1.42	2.46	1.73	0.92	1.54	2.67
Avg.	17649.05	210.94	810.42	4.33	7.68	22.52	83.61

Table 2. Performance compared with vanilla-QEMU and PIN.

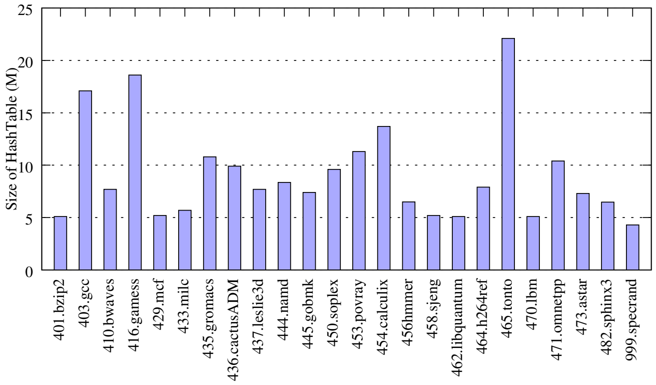


Figure 7. Memory Cost Comparison with SPEC2006 Benchmarks

```
FILE *trace;
VOID SysBefore(ADDRINT ip, ADDRINT num) {
   fprintf(trace,"0x%lx: %ld\n",
          (unsigned long)ip, (long)num);
}
VOID SyscallEntry(THREADID threadIndex,
   CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v) {
   SysBefore(PIN_GetContextReg(ctxt, REG_INST_PTR),
             PIN_GetSyscallNumber(ctxt, std));
}
VOID Fini(INT32 code, VOID *v) {
   printf("program exit()\n");
}
INT32 Usage(VOID){
   return 0;
}
int main(int argc, char * argv[]){
   if(PIN_Init(argc, argv)) return Usage();
   trace = fopen("strace.out", "w");
   PIN_AddSyscallEntryFunction(SyscallEntry, 0);
   PIN_AddFiniFunction(Fini, 0);
   PIN_StartProgram();
   return 0;
}
```

Figure 8. A cross-OS PEMU plugin to trace the syscall.

hostname binaries. We then use PIN and PEMU to analyze the packed hostname.

More specifically, we developed a simple strace (as shown in Fig. 8) plugin to trace the syscall executed by the hostname binary. This plugin will print the syscall number at syscall entry point, and the return value at syscall exit point. We compiled it into a PIN plugin and PEMU plugin with the same source code. PIN failed on these two tests. Both packed programs detected the presence of PIN and exited at early stages. In contrast, hostname ran successfully on PEMU and displayed the host name.

In our other case study, we used eXait [13], a benchmark-like tool to test anti-instrumentation techniques. eXait has a plugin architecture, and each technique is implemented as a separated DLL. There are 21 plugins in total. Again we run PIN with strace plugin to instrument eXait and the loaded DLLs. We found that 17 anti-instrumentation techniques detect the presence of PIN. But none of them detect the presence of PEMU.

Through these case studies, we show there is a need for out-of-VM PIN alternatives. Also, even though future malware will be able to detect the presence of PEMU, we should be able to add countermeasures against them, given that the source code of PEMU is open.

In addition, PEMU can be used to build many out-of-VM introspection tools. In the past several years, we have been using its base internally to build introspection tools such as VMST [14], and EXTERIOR [15]. We believe there will be more use cases of PEMU in this regard.

6. Limitations and Future Work

The current design and implementation of PEMU has a number of limitations. The first one is the incomplete support of the PIN-APIs. Due to the grand engineering challenge, currently we are not able to support all of the PIN-APIs. Besides continuing to finish those unsupported APIs, we would also like to leverage the power from the open source community and make PEMU open source. Being an open source PIN alternative, we believe that there will be more users of PEMU, especially when there is a need to modify the instrumentation engine.

The second limitation is that we used a weak semantic gap [19] when designing the introspection API. That is, while we did not trust any instrumented application code, we did trust the guest OS kernel because we will forward the execution of state inspection related syscalls (e.g., `PEMU_getpid`) to the guest OS. A stronger semantic gap [19] will not trust the guest OS kernel at all. How to retrieve meaningful and trustworthy information from the hypervisor layer when guest OS is untrusted is still an open challenge. One of our future works will investigate how to address this problem.

The third limitation is that we have not attempted to optimize the generated instrumentation and analysis routine yet, though we have designed a number of optimized data structures (e.g., hooking point hash table) to speed up the instrumentation process. Currently, we directly leveraged the optimization from the tiny code generator (TCG) to optimize our instrumentation and analysis routine. We leave the investigation of other optimization techniques such as leveraging parallelism (e.g., [43]) for us to pursue in another future work.

7. Related Work

Over the past 20 years, many dynamic binary instrumentation (DBI) platforms have been developed. In this section, we compare PEMU with these platforms. Note that static binary code instrumentation or rewriting systems, including the first influential link-time instrumentation system ATOM [40], are not within our scope.

At a high level, these dynamic binary instrumentation platforms can be classified into (1) machine simulator, emulator, and virtualizer, (2) process level instrumentation framework, and (3) system wide instrumentation framework. In the following, we discuss these related works and compare PEMU with each of them. A summary of the comparison is presented in Table 3.

Platforms	Year	Emulator, Simulator, Virtualizer	Kernel Level Instrumentation	User Level Instrumentation	w/ API for instrumentation	Out-of-VM	Guest OS Agnostic	PIN API Compatible	Open Source
EMBRA [45]	1996	✓	✓	✓	✗	✓	✗	✗	✗
VMWARE [10]	1998	✓	✓	✓	✗	✓	✗	✗	✗
KERNINST [41]	1999	✗	✓	✓	✓	✗	✓	✗	✓
DYNINSTAPI [7]	2000	✗	✓	✓	✓	✗	✓	✗	✓
DYNAMO [2]	2000	✗	✓	✓	✗	✗	✓	✗	✗
BOCHS [1]	2001	✓	✓	✓	✗	✓	✗	✗	✓
SIMICS [27]	2002	✓	✓	✓	✗	✓	✗	✗	✗
VALGRIND [30, 31]	2003	✗	✗	✓	✓	✗	✓	✗	✓
STRATA [37]	2003	✗	✗	✓	✓	✗	✓	✗	✓
DYNAMORIO [2, 6]	2004	✗	✗	✓	✓	✗	✓	✗	✓
QEMU [4]	2005	✓	✓	✓	✗	✓	✗	✗	✓
PIN [26]	2005	✗	✗	✓	✓	✗	✓	✓	✗
NIRVANA [5]	2006	✗	✗	✓	✓	✗	✓	✗	✗
HDTRANS [39]	2006	✗	✗	✓	✓	✗	✓	✗	✓
VIRTUALBOX [44]	2007	✓	✓	✓	✗	✓	✗	✗	✓
PINOS [8]	2007	✓	✓	✓	✓	✗	✗	✓	✗
TEMU [48]	2010	✓	✓	✓	✓	✓	✓	✗	✓
DYNINST [28]	2010	✗	✓	✓	✓	✗	✓	✗	✓
DRK [34]	2013	✗	✓	✓	✓	✗	✓	✗	✓
DECAF [17]	2014	✓	✓	✓	✓	✓	✗	✗	✓
PEMU	2015	✓	✓	✓	✓	✓	✓	✓	✓

Table 3. Comparison with other dynamic binary instrumentation platforms.

Machine Simulator, Emulator, and Virtualizer. The very early development of dynamic binary code instrumentation originated from machine simulation or emulation. EMBRA [45] is such a simulation system. It performs whole system dynamic translation for MIPS architectures. BOCHS [1] and SIMICS [27] are also simulators that allow the instrumentation and inspection of all the executed x86 instructions. Targeting x86 architecture, the very early versions of VMWARE [10] also use dynamic binary translation to build virtual machine monitors (VMM). Another widely used VMM or emulator is QEMU [4], which supports a large number of architectures. When used as an emulator, QEMU uses a tiny code generator to emulate a CPU through a binary translation. QEMU can also be used as a virtualizer recently, and it can cooperate with the Xen hypervisor or KVM kernel module to achieve a near native performance through running the guest code directly on host CPU. VIRTUALBOX [44] is a faster VMM compared to QEMU, and it employs an in-situ patching to achieve better performance.

For all these out-of-VM works, they certainly can instrument both user level and kernel level code, but they do not offer any APIs for users to build dynamic binary instrumentation applications. Also, when used to analyze guest kernels, they all tend to be kernel specific.

Process Level Instrumentation Framework. Recognizing the importance and wide applications of DBI, many process level instrumentation frameworks (e.g., DYNINSTAPI [7], STRATA [37], DYNAMORIO [2], VALGRIND [30], PIN [26], NIRVANA [5], HDTRANS [39], DYNINST [28]) have been proposed. These frameworks offer APIs for developers to build plugins for various applications such as high performance simulation [37], program shepherding [23], and memory error detection [38].

Among them, VALGRIND [30, 31] is a comprehensive DBI framework that offers rich APIs for dynamic binary instrumentation. It supports various architectures (e.g., x86, ARM, MIPS) due to the use of an intermediate representation (IR) that is processor-neutral and SSA-based. Similar to VALGRIND, PIN [26] also works at user space, but it only supports IA-32 and x86-64 architectures. Developers can create PIN-tools using the APIs provided by PIN, and execute them atop either Windows or Linux. It is featured with 'ease of use' with rich APIs to abstract away the underlying instruction-set idiosyncrasies. Making PIN-API compatible is one of our design goals such that PIN users can easily switch to our platform, especially when there is a need to customize the underlying DBI engine. Unlike other DBI platforms, DYNINST [28] can instrument at any time in the execution of a program, from static instrumentation (i.e., binary rewriting) to dynamic instrumentation (i.e., instrumenting actively while executing the code). Also, it allows users to modify or remove instrumentation at any time, with such modifications taking immediate effect.

For process level instrumentation, they are efficient. They are built atop OS, and thus are OS-agnostic. It is also easier to develop the plugins. However, the analysis routine and the original program code share the same address space. Therefore, they are all in-VM approaches, and users have to be cautious when applying them for security sensitive applications.

System Wide Instrumentation Framework. In addition to process level instrumentation, there is also a need for OS kernel instrumentation. KERNINST [41], PINOS [8], TEMU [48], DRK [34], and DECAF [17] are such systems.

Among them, KERNINST and DRK are built atop in kernel dynamic binary code translation. They basically control all kernel instruction execution, and enable comprehensive instrumentation of the OS kernel code. PINOS [8] is a whole-system instrumentation extension of PIN. It takes advantage of Intel VT Technology to interpose between the subject system and hardware. PINOS has been implemented based on the Xen virtual machine monitor. Compared to PEMU, the instrumentation and analysis code of KERNINST, PINOS and DRK actually share the same address space. Even though PINOS steals the memory from the guest OS, the monitored process is still able to guess and access the memory used by analysis routines. Therefore, they do not offer strong out-of-VM isolation.

TEMU [48] is a whole-system instrumentation tool built atop QEMU. A unique feature in TEMU is that it offers APIs for dynamic taint analysis and in-depth program behavioral analysis. It is an out-of-VM based instrumentation, but it installed a helper kernel module inside the guest OS to report the states to the outside analysis routine. The most recent effort, DECAF, extends TEMU. It does not use any in-VM kernel module anymore, but the way to bridge the semantic gap still requires the knowledge of kernel data structures. Therefore, DECAF is a more OS-specific solution. For TEMU, it is less since it is an in-VM based approach.

8. Conclusion

We have presented the design, implementation, and evaluation of PEMU, a new dynamic binary code instrumentation framework that allows end-users to develop out-of-VM plugins for various program analyses. One distinctive feature of PEMU is its PIN-API compatibility. Therefore, many of the PIN-tools can be recompiled and executed within our framework. Unlike other similar systems, it is guest-OS agnostic, and can execute many different guest OSes with different versions. Our experimental results with SPEC 2006 benchmarks show that PEMU introduces reasonable overhead.

Acknowledgement and Availability

We thank the anonymous reviewers for their insightful comments. We are also grateful to Erick Bauman for his invaluable feedback on an early draft of this paper. This research was supported in part by a AFOSR grant FA9550-14-1-0119 and a DARPA grant 12011593. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the AFOSR and DARPA. Finally, the source code of PEMU is available at https://github.com/utds3lab/pemu.

References

[1] bochs: The open source ia-32 emulation project, 2001. http://bochs.sourceforge.net/.

[2] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 1–12.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUERY, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003).

[4] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association.

[5] Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., and Chau, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 154–163.

[6] Bruening, D., Zhao, Q., and Amarasinghe, S. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (New York, NY, USA, 2012), VEE '12, ACM, pp. 133–144.

[7] Buck, B., and Hollingsworth, J. K. An api for runtime code patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (Nov. 2000), 317–329.

[8] Bungale, P. P., and Luk, C.-K. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), pp. 137–147.

[9] Chen, P. M., and Noble, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (2001), pp. 133–138.

[10] Devine, S. W., Bugnion, E., and Rosenblum, M. Virtualization System Including a Virtual Machine Monitor for a Computer with a Segmented Architecture. *United States Patent 6,397,242* (1998).

[11] Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., and Lee, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy* (Oakland, CA, USA, 2011), pp. 297–312.

[12] Egele, M., Kruegel, C., Kirda, E., Yin, H., and Song, D. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), ATC'07, USENIX Association, pp. 18:1–18:14.

[13] Francisco Falcã̧n, N. R. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *recon* (2012).

[14] Fu, Y., and Lin, Z. Space traveling across vm: Automatically bridging the semantic-gap in virtual machine introspection via online kernel data redirection. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2012).

[15] Fu, Y., and Lin, Z. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments* (Houston, TX, March 2013).

[16] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Sym. (NDSS'03)* (February 2003).

[17] Henderson, A., Prakash, A., Yan, L. K., Hu, X., Wang, X., Zhou, R., and Yin, H. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2014), ISSTA 2014, ACM, pp. 248–258.

[18] Hofmann, O. S., Dunn, A. M., Kim, S., Roy, I., and Witchel, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems* (Newport Beach, California, USA, 2011), ASPLOS '11, pp. 279–290.

[19] Jain, B., Baig, M. B., Zhang, D., Porter, D. E., and Sion, R. Sok: Introspections on trust and the semantic gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2014), SP '14, IEEE Computer Society, pp. 605–620.

[20] Jiang, X., Wang, X., and Xu, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)* (Alexandria, Virginia, USA, 2007), ACM, pp. 128–138.

[21] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Antfarm: tracking processes in a virtual machine environment. In *Proc. annual Conf. USENIX '06 Annual Technical Conf.* (Boston, MA, 2006), USENIX Association.

[22] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Vmm-based hidden process detection and identification using lycosid. In *Proc. fourth ACM SIGPLAN/SIGOPS international Conf. Virtual execution environments* (Seattle, WA, USA, 2008), VEE '08, ACM, pp. 91–100.

[23] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[24] Lin, Z., Zhang, X., and Xu, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)* (San Diego, CA, February 2010).

[25] Lu, S., Tucek, J., Qin, F., and Zhou, Y. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ASPLOS XII, ACM, pp. 37–48.

[26] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[27] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B. Simics: A full system simulation platform. *Computer 35*, 2 (Feb. 2002), 50–58.

[28] Miller, B. P., and Bernat, A. R. Anywhere, any time binary instrumentation.

[29] Narayanasamy, S., Pereira, C., Patil, H., Cohn, R., and Calder, B. Automatic logging of operating system

effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2006), SIGMETRICS '06/Performance '06, ACM, pp. 216–227.

[30] NETHERCOTE, N., AND SEWARD, J. Valgrind: A program supervision framework. In *In Third Workshop on Runtime Verification (RV'03)* (2003).

[31] NETHERCOTE, N., AND SEWARD, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[32] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed Systems Security Symposium* (2005).

[33] PAYNE, B. D., CARBONE, M., AND LEE, W. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)* (December 2007).

[34] PETER FEINER, A. D. B., AND GOEL, A. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems* (2012).

[35] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the 14th ACM conference on Computer and communications security* (2007), CCS '07, pp. 103–115.

[36] SCHWARTZ, E. J., AVGERINOS, T., AND BRUMLEY, D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 317–331.

[37] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2003), CGO '03, IEEE Computer Society, pp. 36–47.

[38] SEWARD, J., AND NETHERCOTE, N. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association.

[39] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. Hdtrans: An open source, low-level dynamic instrumentation system. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 175–185.

[40] SRIVASTAVA, A., AND EUSTACE, A. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1994), PLDI '94, ACM, pp. 196–205.

[41] TAMCHES, A., AND MILLER, B. P. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 117–130.

[42] WALLACE, S., AND HAZELWOOD, K. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Annual International Symposium on Code Generation and Optimization* (San Jose, CA, March 2007), pp. 209–217.

[43] WANG, Z., LIU, R., CHEN, Y., WU, X., CHEN, H., ZHANG, W., AND ZANG, B. Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2011), PPoPP '11, ACM, pp. 213–222.

[44] WATSON, J. Virtualbox: Bits and bytes masquerading as machines. *Linux J. 2008*, 166 (Feb. 2008).

[45] WITCHEL, E., AND ROSENBLUM, M. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1996), SIGMETRICS '96, ACM, pp. 68–79.

[46] WU, Q., REDDI, V., WU, Y., LEE, J., CONNORS, D., BROOKS, D., MARTONOSI, M., AND CLARK, D. A dynamic compilation framework for controlling microprocessor energy and performance. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on* (2005).

[47] YANGCHUN FU, J. Z., AND LIN, Z. Hypershell: A practical hypervisor layer guest os shell for automated in-vm management. In *USENIX ATC'14 Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (USENIX Association Berkeley, CA, USA, 2014), USENIX Association, pp. 85–96.

[48] YIN, H., AND SONG, D. Temu: Binary code analysis via whole-system layered annotative execution. Technical Report UCB/EECS-2010-3, EECS Department, University of California, Berkeley, Jan 2010.

[49] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), CCS '07, ACM, pp. 116–127.