Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform

Andrew Henderson*, Aravind Prakash*, Lok Kwong Yan†, Xunchao Hu*,
Xujiewen Wang*, Rundong Zhou*, and Heng Yin*
*Department of EECS, Syracuse University, Syracuse, New York, USA
†Air Force Research Lab, Rome, New York, USA
{anhender,arprakas,loyan,xhu31,xwang166,rzhou02,heyin}@syr.edu

ABSTRACT

Dynamic binary analysis is a prevalent and indispensable technique in program analysis. While several dynamic binary analysis tools and frameworks have been proposed, all suffer from one or more of: prohibitive performance degradation, semantic gap between the analysis code and the program being analyzed, architecture/OS specificity, being user-mode only, lacking APIs, etc. We present DECAF, a virtual machine based, multi-target, whole-system dynamic binary analysis framework built on top of QEMU. DECAF provides Just-In-Time Virtual Machine Introspection combined with a novel TCG instruction-level tainting at bit granularity, backed by a plugin based, simple-to-use event driven programming interface. DECAF exercises fine control over the TCG instructions to accomplish on-the-fly optimizations. We present 3 platform-neutral plugins - Instruction Tracer, Keylogger Detector, and API Tracer, to demonstrate the ease of use and effectiveness of DECAF in writing cross-platform and system-wide analysis tools. Implementation of DECAF consists of 9550 lines of C++ code and 10270 lines of C code and we evaluate DECAF using CPU2006 SPEC benchmarks and show average overhead of 605% for system wide tainting and 12% for VMI.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Security

Keywords

Dynamic binary analysis, dynamic taint analysis, virtual machine introspection

1. INTRODUCTION

Dynamic binary analysis has demonstrated its strength in many research problems, such as malware analysis, protocol reverse engineering, vulnerability signature generation, software testing, profiling and performance optimization, etc. There are many analysis platforms for process-level binary instrumentation, such as Pin [15] and Valgrind [16].

Compared to process-level program instrumentation and analysis, whole-system dynamic binary analysis has its unique advantages. First, it provides a full system view, including the OS kernel and all running applications. So, we can analyze kernel activity and the interactions among multiple processes. Second, the code instrumentation and analysis are performed completely from outside of the virtual machine (VM). In contrast, user-level instrumentation tools share the same memory space as the instrumented program execution. Whole-system dynamic binary analysis provides better transparency and stronger isolation than that of process-level instrumentation tools. This is especially important in the context of analyzing malicious code that attempts to detect, evade, and tamper with the analysis environment.

Although much research has been done to make use of whole-system dynamic binary analysis to solve various security problems [3, 18, 6, 4, 25, 26], little attention has been paid to the analysis framework itself. Such tools are often tailored to solve specific problems in ad-hoc manners. Many times, analysts still must develop new analysis tools from scratch to meet their own needs.

Building a generic, whole-system dynamic binary analysis platform that suits various needs is desirable, but challenging. For example, previous work in this area, TEMU [21] in the BitBlaze binary analysis toolkit [20], provides a rich set of capabilities and has facilitated many binary analysis research projects. However, many of its design and implementation choices are fairly ad-hoc and cumbersome. So, it can only "Make It Work". From time to time, TEMU falls short in analysis capability, correctness, and efficiency.

In this paper, we present *DECAF*[1], a new whole-system dynamic binary analysis platform that aims to "Make It Work, Make It Right, Make It Fast". This means that DECAF must not only provide the same set of capabilities as TEMU, but it must follow the proper principles in its design. DECAF offers analysis results of better quality, and with a higher correctness guarantee, than TEMU while still conducting analyses more efficiently. Particularly, in DECAF, we overcome the following key challenges in building a whole-system dynamic binary analysis platform:

[1]DECAF stands for Dynamic Executable Code Analysis Framework.

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610407

1. How to reconstruct fresh OS-level semantic view completely from outside? As we run a virtual machine inside a whole-system binary analysis framework and perform various analysis tasks from outside, we have to reconstruct the OS-level semantic view of the virtual machine from outside, which is called Virtual Machine Introspection (VMI). Several efforts (such as VMWatcher [22], Virtuoso [9], and VMST [10]) have been made to bridge the semantic gap. These systems have answered how to reconstruct the OS-level semantic view. However, the question of "when to reconstruct" is not addressed. In a running system, the OS-level semantic views keep changing (e.g., a process starts or terminates, a code module is loaded or unloaded). For dynamic analysis, we need to know these new events "just-in-time". TEMU circumvented this problem by inserting a kernel module into the guest OS within the VM. This kernel module hooks several system events, retrieves the OS-level information, and passes it to the hypervisor through a spare port. This circumvention clearly violates the external monitoring principle for VMI and can be easily subverted by the malicious code inside the VM. In DECAF, we find a better solution to reconstructing fresh OS-level semantic view by only monitoring hardware-level events.

2. How to provide an event-based programming paradigm that is both correct and efficient? Most of the existing analysis platforms provide instrumentation interfaces only, by which a plugin can specify which instructions to instrument and what instrumentation code should run. While this instrumentation interface is simple and flexible, it leaves the burden to the plugin developers to decide how to instrument the program execution. While this approach is acceptable for user-level instrumentation, it is difficult for a whole-system setting, because in order to know how to properly instrument the whole-system execution, the analyst must be familiar with low-level system details, such as exceptions, interrupts, page faults, context switches, etc. Therefore, DECAF provides an event-based interface, through which the analyst can register for events in the selected contexts (e.g., a process, the kernel space, or a kernel module). Under the hood, DECAF takes care of what instrumentation code to insert and where, and it ensures that the inserted instrumentation code is correct and efficient. TEMU was also aimed to provide the same high-level interface, but achieved it in a naive way: it inserts instrumentation code uniformly in all the translated code blocks and decides at execution time whether to deliver the events to the plugin. It guarantees the correctness of event processing, but incurs unnecessarily high runtime overhead.

3. How to implement precise and lossless tainting? Dynamic taint analysis (tainting) is a powerful dynamic binary analysis technique. There have existed many implementations [20, 5, 17, 19, 2]. Among all these implementations, two important factors are overlooked. Most of these implementations are not precise enough, and some of them are not even sound. This means that these taint analysis systems would unnecessarily mark many memory locations as tainted (overtainting) and also fail to taint certain memory locations and CPU registers that are supposed to be tainted (undertainting). Moreover, often times, we need to keep track of tainted data from multiple taint sources (labels). Many taint analysis implementations do not distinguish among multiple taint labels. For the ones that do keep track of multiple taint labels, they do not provide a lossless guarantee. Each tainted byte or word is associated with up to a small number of tainted labels, due to the space constraint of the shadow memory. When a memory location or CPU register is tainted from more tainted sources than those that can be kept in the shadow memory, the rest are *lost*! To achieve high precision, DECAF maintains bit-level precision taint information for every bit of registers and memory locations and applies precise tainting rules for most instructions at the intermediate representation level. To achieve the lossless requirement without sacrificing efficiency, DECAF pushes the taint label propagation off the main code execution stream, and keeps track of taint labels in an asynchronous manner via detailed logging using its plugins.

4. How to provide strong support for cross-platform analysis? Ideally, we would like to have the same analysis code (with minimum platform-specific code) to work for different CPU architectures (e.g, x86 and ARM) and different operating systems (e.g., Windows and Linux). It requires that the analysis framework hide the architecture and operating system specific details from the analysis plugins. Further, to make the analysis framework itself maintainable and extensible to new architectures and OSes, the platform-specific code within the framework should also be minimized. Note that this cross-platform support includes both the architecture of the VM and the OS running within the VM. Some instrumentation tools, like Pin [15], can run in both Linux and Windows, but, until now, no analysis tools can provide support for both multiple architectures and multiple OSes. DECAF provides support for multiple platforms by implementing core instrumentation and analysis tasks in the intermediate representation layer (independent of the CPU architecture of the VM). DECAF's plugin API is engineered to hide many architecture and OS specific details.

DECAF is an open-source project [7]. Since the release of its first version in January 2013, it has received over 3500 downloads. A handful of analysis plugins have also been built on top of it to demonstrate the power of this framework. We showcase three plugins to demonstrate how DECAF enables and solves various binary analysis problems. By hooking the entries and exits of APIs specified in a configuration file, *API Tracer* is able to trace the API invocations of a specified process and the processes spawned from it. *Keylogger Detector* tracks tainted keystrokes propagating throughout the OS kernel and across user-level processes to detect keyloggers. *Instruction Tracer* logs instructions executing within a specified context (such as a user-level process, or a kernel module). These plugins are mostly platform neutral. Since DECAF provides a platform-independent programming interface, these plugins can analyze binary executables for multiple hardware architectures (including x86 and ARM) and multiple OSes (including Windows and Linux), requiring no, or very minimal, platform-specific code.

2. SYSTEM OVERVIEW

Generally speaking, a virtual machine is running on top of DECAF (an "enhanced" QEMU [1]). QEMU makes use of dynamic binary translation techniques to emulate multiple target architectures.

2.1 Architecture

Figure 1 illustrates the overall architecture of DECAF. Inside the virtual machine, we can run the programs of interest and conduct various analyses externally via analysis

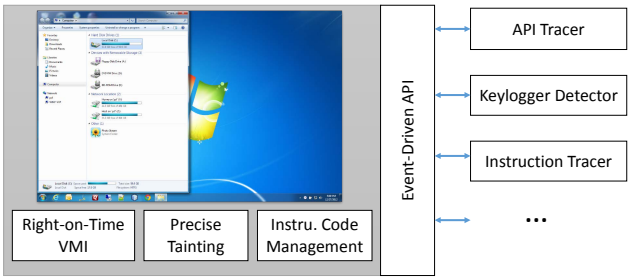

Figure 1: The overall architecture of DECAF.

plugins. To provide various analysis capabilities, DECAF gets involved extensively with the dynamic binary translation process, which is detailed in Section 3. DECAF has the following key components:

Just-In-Time VMI. This VMI component is able to reconstruct a fresh OS-level view of the virtual machine, including processes, threads, code modules, and symbols, to support binary analysis. Further, to support multiple architectures and operating systems, it follows a platform-neutral design principle. The workflow for extracting OS-level semantic information is common across multiple architectures and OSes. The only platform-specific handling lies in what kernel data structures and what fields to extract information from. We present more details about VMI in Section 4.

Precise, lossless dynamic taint analysis. DECAF ensures precise tainting by maintaining bit-level taint precision for CPU registers and memory, and inlining precise tainting rules in the translated code blocks. Thus, the taint status for each CPU register and memory location is processed and updated synchronously during the code execution of the virtual machine. The propagation of taint labels is done in an asynchronous manner for two reasons: 1) it is impractical and expensive to maintain unlimited amount of labels for each tainted bit in the shadow memory; and 2) for most taint analysis problems, we do not need to know tainted labels for all tainted bits in real time. We are only interested in a small amount of tainted data (e.g., tainted EIP or a tainted buffer), and when they become tainted, we can then trace back through the taint propagation log and retrieve their labels. By implementing such a tainting logic mainly in the intermediate representation level (more concretely, TCG instruction level), it becomes easy to extend tainting support to a new CPU architecture. Section 5 provides more details about our taint analysis implementation.

Event-driven programming interface. Compared to many of existing analysis frameworks [15, 16] that provide just the instrumentation interface, DECAF provides an event-driven programming interface. It means that the paradigm of "instrument in the translation phase and then analyze in the execution phase" is invisible to the analysis plugins. The analysis plugins only need to register for specific events and implement the corresponding event handling functions. The details of code instrumentation are taken care of by the framework. Such details include how to generate the instrumentation code for inserting these event handlers into the translated code stream and how to maintain instrumentation code consistency when new event handlers are registered and old ones are removed.

Dynamic instrumentation management. To reduce runtime overhead, the instrumentation code is inserted into the translated code only where necessary. For example, when a plugin registers a function hook for a function's entry point, the instrumentation code for this hook is only placed once (at the function entry point). When the plugin unregisters this function hook, the instrumentation code will also be removed from the translated code accordingly. To ease the development of plugins, the management of dynamic code instrumentation is completely taken care of in the framework, and thus invisible to the plugins.

```
1. plugin_interface_t my_interface;
2. DECAF_Handle keystroke_cb_handle = DECAF_NULL_HANDLE;
3. DECAF_Handle handle_read_taint_mem = DECAF_NULL_HANDLE;
4. int taint_key_enabled = 0;

5. void my_read_taint_mem(DECAF_Callback_Params *param) {
6.  char name[128];
7.  tmodinfo_t tm;
8.  if(VMI_locate_module_c(DECAF_getPC(cpu_single_env),
      DECAF_getPGD(cpu_single_env),name,&tm) == 0)
9.    DECAF_printf("INSN 0x%08x From Module %s Read Keystroke\n",
            DECAF_getPC(cpu_single_env),tm.name);
   }
10. void my_send_keystroke_cb(DECAF_Callback_Params *params) {
11.  *params->ks.taint_mark = taint_key_enabled;
12.  taint_key_enabled = 0;
13.  DECAF_printf("taint keystroke %d \n", params->ks.keycode);
    }
14. void do_taint_sendkey(Monitor *mon,const QDict *qdict) {
15.  if (qdict_haskey(qdict, "key")) {
16.   taint_key_enabled = 1; //enable keystroke taint
17.   do_send_key(qdict_get_str(qdict, "key")); //Send the key
     }
    }
18. mon_cmd_t my_term_cmds[] = {
     {
19.   .name  = "taint_sendkey",
20.   .args_type = "key:s",
21.   .mhandler.cmd = do_taint_sendkey,
22.   .params = "taint_sendkey key",
23.   .help = "send a tainted key to system"
     },
     {NULL, NULL, },
    };
24. void my_cleanup(){......}
  /* Register the plugin and the callbacks */
25. plugin_interface_t * init_plugin() {
26.  my_interface.mon_cmds = my_term_cmds;
27.  my_interface.plugin_cleanup = my_cleanup;
28.  handle_read_taint_mem = DECAF_register_callback(
          DECAF_READ_TAINTMEM_CB, my_read_taint_mem, NULL);
29.  keystroke_cb_handle = DECAF_register_callback(
          DECAF_KEYSTROKE_CB, my_send_keystroke, NULL);
30.  return &keystroketaint_interface;
    }
```

Figure 2: A sample plugin that keeps track of tainted keystrokes

2.2 Sample Plugin

Figure 2 presents a sample plugin that detects keyloggers by tracking the propagation of tainted keystrokes throughout the entire guest environment. When this plugin is loaded into the analysis framework, its `init_plugin` function is called to initialize the plugin and return a pointer to `plugin_interface_t`, which specifies a new terminal command (`taint_sendkey`) defined by the plugin and a plugin cleanup function. In addition, the plugin registers two callbacks, one (`my_read_tainted_mem`) for a tainted memory read, and the other (`my_send_keystroke`) for sending a keystroke.

When the user enters the `taint_sendkey` command in the terminal, the registered callback `my_send_keystroke` is called and the corresponding keystroke is tainted. Thereafter, the tainted keystroke will propagate from the keyboard device, through the OS kernel, and to the destination user-level program. Since DECAF performs whole-system dynamic taint analysis, we are able to observe this entire taint propagation flow. Whenever an instruction reads a

tainted memory location, the framework will call the registered `my_read_tainted_mem` callback, which checks the code module in which this instruction is located, thanks to our Just-In-Time VMI support. The relevant information about this taint event is logged for offline analysis.

It is worth noting that this sample plugin is platform and OS independent. The same plugin code works for x86 and ARM, Windows and Linux. Whenever possible, DECAF provides generic functions to access architecture-dependent features. For example, `DECAF_getPC` will return the program counter (e.g., `EIP` in x86 and `R15` in ARM), and `DECAF_getPGD` will return the page table directory (e.g., `CR3` in x86 and `CP15` in ARM).

3. SELECTIVE CODE INSTRUMENTATION

To meet the requirements of efficiency and cross-platform for code instrumentation, DECAF selectively inserts instrumentation code at the intermediate representation level.

Dynamic binary translation in QEMU. To support multiple architectures, QEMU makes use of a compiler backend, called Tiny Code Generator (TCG), as its dynamic binary translation engine. QEMU translates each basic block of guest instructions into a series of architecture-independent TCG instructions grouped together as a TCG translation block (TB). The TCG compiler translates each TB into a piece of native code to be executed on the host. Figure 3(a) shows how two x86 instructions are translated into TCG instructions. TCG instructions include common ALU operations (e.g. `add`, `sub`, `xor`), memory load/store, and control flow transfer. The parameters for each TCG instruction can be temporary variables, global variables, and constants. For more complex, guest-specific instructions (e.g. floating point operations), a `call` TCG instruction exists for making calls to high-level language helper functions that implement the complex functionality. In this manner, TCG cleanly decouples specific details of the guest from that of the host. Our code instrumentation must work coherently with the TCG-based dynamic binary translation process.

Placement of code execution events. Events like "block begin/end" and "instruction begin/end" are used for tracing program execution. When callbacks for these events are registered by a plugin, DECAF inserts the proper helper function calls into the necessary TBs by pausing the guest's execution, flushing the necessary TBs, retranslating those TBs to include calls to the helper functions, and then resuming the guest's execution. Since callbacks are triggered inline with the guest's execution, they are synchronized to the occurance of events of interest. Figure 3(b) shows that the two helper functions `DECAF_invoke_insn_begin_callback` and `DECAF_invoke_insn_end_callback` are inserted at the beginning and the end of each guest instruction. For many analyses, we are only interested in the execution of a small portion of the system, such as a single kernel module or user-level process. Plugins can specify ranges of memory addresses, or even a single address, of interest when registering for callbacks. Callback helper functions are placed into only the necessary TBs, and only at the proper locations within each TB, to capture these events as they occur.

An important design decision here is a dispatch mechanism. For each kind of event (e.g., "block begin"), we only insert a single helper function (e.g., `DECAF_invoke_block_begin_callback`) at each desired program location, and within the helper function, we will iterate through all the registered callbacks for that event and decide which callbacks to trigger. There are two important reasons. The plugins and the platform itself may altogether register multiple callbacks on the same event. A dispatch mechanism like this can avoid inlining repeated helper function calls, which negatively impacts the performance. More importantly, in this whole-system emulator, the callback functions inserted into the code stream are executed within the context of the entire guest system. For example, instrumentation code inserted into a shared library will be executed in all processes with this library loaded. So, we need the dispatch mechanism to decide at execution time if the current execution context is the correct one for each registered callback.

```
// Start of translation block
// Original instruction: orl  %ebx, %eax
mov_i32  tmp11, ebx
mov_i32  tmp12, eax
or_i32   tmp13, tmp12, tmp11
// Original instruction: addl $0x01, %eax
movi_i32 tmp14, $0x01
add_i32  tmp15, tmp14, tmp13
mov_i32  eax, tmp15
// End of translation block
goto_tb  $0x0
```

(a)

```
// Start of translation block
// Insert DECAF_BLOCK_BEGIN callback
movi_i32  tmp21, $<CURRENT_ADDRESS>
movi_i32  tmp22, $DECAF_invoke_block_begin_callback
call      tmp22, $0x0, $0, env, tmp21
// Original instruction: orl  %ebx, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32  tmp23, $DECAF_invoke_insn_begin_callback
call      tmp23, $0x0, $0, env
mov_i32   tmp11, ebx
mov_i32   tmp12, eax
or_i32    tmp13, tmp12, tmp11
// Insert DECAF_INSN_END callback
movi_i32  tmp24, $DECAF_invoke_insn_end_callback
call      tmp24, $0x0, $0, env
// Original instruction: addl $0x01, %eax
// Insert DECAF_INSN_BEGIN callback
movi_i32  tmp25, $DECAF_invoke_insn_begin_callback
call      tmp25, $0x0, $0, env
movi_i32  tmp14, $0x01
add_i32   tmp15, tmp14, tmp13
mov_i32   eax, tmp15
// Insert DECAF_INSN_END callback
movi_i32  tmp26, $DECAF_invoke_insn_end_callback
call      tmp26, $0x0, $0, env
// End of translation block
// Insert DECAF_BLOCK_END callback
movi_i32  tmp27, $DECAF_invoke_block_end_callback
call      tmp27, $0x0, $0, env
goto_tb   $0x0
```

(b)

Figure 3: DECAF inserts instruction execution callbacks into the original TCG code stream (a) to create an instrumented opcode stream (b) to trigger helper function calls to plugin callback functions.

We also need a mechanism to nicely remove any stale instrumentation code. A plugin may frequently register and remove callbacks at runtime. A common example is function hooking. A plugin may need to examine the return value and output parameters when an API call returns. To do so, the plugin registers a hook on the entrypoint of that call. When that hook is invoked, the plugin retrieves the return address of the API call and registers a second hook on its return address. When the second hook is invoked, the plugin can inspect the return value and output parameters. After that, the plugin removes the second hook for efficiency. Thanks to the dispatch mechanism described above, we no longer have to immediately remove the second hook , which involves flushing the corresponding code cache and forcing a retranslation, which hurts runtime performance. If no callbacks are associated with an inserted helper function, then no callbacks will be dispatched, which is expected. This little extra function call overhead is several magnitudes smaller than frequent code cache flushing and retranslation. Therefore, we postpone the actual code cache flush to a much later time to improve efficiency. All these are done under the hood by DECAF.

MMU, IO, and higher-level events. Events like "memory read/write" and "tainted memory read/write" are related to the Software Memory Management Unit (in short, SOFTMMU) in QEMU. QEMU must translate each guest virtual

address into a guest physical address, and then translate that into a host virtual address. Therefore, the instrumentation for MMU-related events is straightforward: the helper functions are directly inserted in the SOFTMMU code. Of course, a dispatch mechanism is still needed to properly deliver the callbacks to the plugin. Some higher-level events are derived from these low-level memory events. For example, VMI events (such as process creation and deletion) are derived from the TLB execute miss event.

QEMU emulates a set of common IO devices, such as hard disks, keyboards, and network cards. We can easily instrument the IO events related to these devices by inserting helper functions inside each virtual device's implementation.

Dynamic tainting control. A unique feature of DECAF is that it can switch tainting on and off dynamically. This is particularly important for a whole-system analysis framework. Due to the considerable runtime overhead of tainting, we would like to start tainting support only when it is needed. When a user or plugin requests to switch tainting on or off, DECAF will flush the entire translation code cache and instrument the new code blocks under the new settings. Details of the implementation of tainting instrumentation at the TCG-instruction level are explained in Section 5.

4. JUST-IN-TIME VMI

As a binary analysis platform, DECAF needs to reconstruct the following OS-level semantic knowledge of the VM to facilitate custom analysis tasks "out of the box": (1) **Process.** We need to know what processes are running in the VM. As many analysis tasks only focus on one or several user-level processes, the process information is essential. (2) **Thread.** Many programs are multi-threaded. Knowing what threads are running within a given process is also important for many analysis tasks. (3) **Code modules.** Within a process's memory space, a main executable and several shared libraries are loaded. Binary analysis often needs to know which code module an instruction comes from. Thus, this code module information is often required. (4) **Exported symbols.** Shared libraries export a list of functions, such that the other code modules can dynamically link with each other and call these exported functions by name. Retrieving these exported symbols can greatly help in understanding a program's behavior at the API level (because APIs are exported symbols).

4.1 Goals and Challenges

We have the following design goals for VMI. First of all, we would like to obtain a fresh view of the guest OS. For many analysis tasks, we need to know immediately when a new process is created or a new code module is loaded so that we can observe a program's complete execution from beginning to end. No existing VMI techniques provide such a strong timing guarantee. In addition, we would like our VMI technique to be as platform-independent as possible, as the same technique should work for different CPU architectures and different OSes with minimal platform-specific handling. Note that to achieve such a strong timing guarantee, one could hook specific system calls (e.g., `fork` and `exec`) or kernel functions. However, this approach is very OS-specific and often changes across different OS versions. Last but not least, as a basic functionality required by almost every analysis plugin, the performance overhead for our VMI technique should be minimal. A key challenge for our VMI technique is to meet both this performance requirement and the strong timing guarantee simultaneously because we have to monitor certain system events more frequently, which may incur high runtime overhead, to obtain a fresh view of the guest OS.

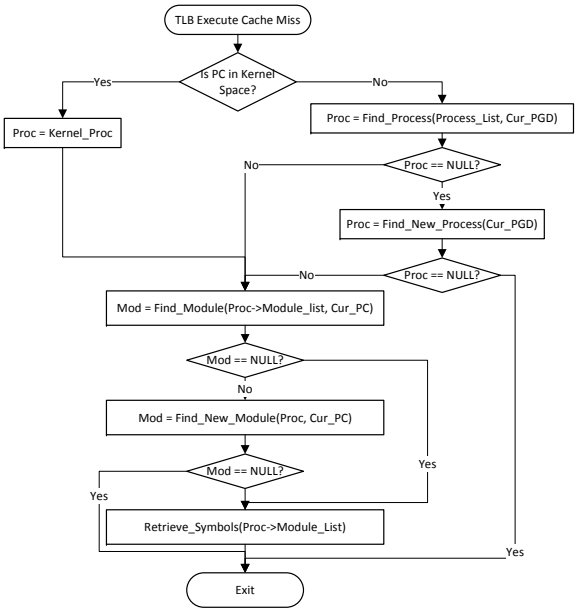

Figure 4: The VMI flowchart

4.2 Solution

We rely on the following observations that commonly hold true across platforms to achieve just-in-time VMI. First of all, a process must have its own memory space and each CPU architecture must have a register to indicate the current memory space (e.g., `CR3` in x86 and `CP15` in ARM), so we can use this register to identify a new process. Second, a Translation Look-aside Buffer (TLB) will have an "execute" cache miss whenever a new code page is loaded and executed. Third, upon context switch, the old mappings in the TLB will be flushed. Therefore, our VMI is triggered by TLB Execute cache misses, because whenever a new process is created or a new module is loaded, we must capture the right moment on a TLB Execute cache miss.

Figure 4 illustrates this workflow. Whenever we observe a TLB Execute cache miss, we will first check if the current program counter is in the kernel space. If not, we will check if the current process is newly created by searching the current PGD in the process list. If we cannot find it, this process must be new, so we will traverse the kernel data structures (i.e., active process list) in the VM to retrieve information about the newly created process. As you can see, we only traverse kernel data structures (which can be a costly operation), when there is a new process. Checking the existing processes in the hash table takes constant time.

After we locate the right process (either it is already existing or newly created), we would check if a new code module is loaded. Again, we have a hash table to quickly check if the current program counter falls into any code modules that have been loaded into the current process memory space. If not, we find a new code module, and we traverse the module list in the VM to retrieve the information (such as module name, base address, size) about the new module.

After we locate the current code module (either it is already loaded or new), we then start to retrieve the exported symbols of the code modules directly from memory. We

will need to parse the headers (PE for Windows, and ELF for Linux) of each code module to extract symbols. Note that we may not be able to completely retrieve symbols for a newly loaded module the first time we see it, because the related pages may not be loaded in the memory at that time. Therefore, on future TLB Execute misses, we will check this code module to see if the symbols are available to retrieve.

The symbol extraction process sounds fairly heavyweight because it requires many memory reads from the VM to parse executable headers and copy the symbols. However, we only need to do it once for each code module across all the processes. Since most code modules are shared libraries, like .so files in Linux and .dll files in Windows, this overhead is amortized across the creation of multiple processes.

TLB cache misses cannot help us find the exact moment when a process has terminated or a module has unloaded. To find such events, we will periodically traverse the kernel data structures to find the deleted process objects and unloaded code modules. In general, these events are not so timing critical for binary analysis purposes, unlike process creation and module loading events. So, periodically checking (e.g., every 1 or 5 seconds) is acceptable. If for certain problems we do need to know precise time when such events happen, the plugins would need to implement their own mechanism, such as hooking specific functions in the guest execution.

As we can see, this VMI workflow avoids inserting OS-specific hooks into the VM to obtain a fresh view of the guest OS, and it also avoids frequent memory reads in the VM. The only platform-specific knowledge for this VMI workflow is what kernel data structures to examine and how to interpret the related fields in these kernel data structures. The definition of these data structures are publically available. Compared to hooking into system calls and kernel functions, this approach is more stable. Changes on kernel data structures are less frequent than code. It is also fairly straightforward to extract the data structure information from the public symbols of guest OSes.

5. PRECISE LOSSLESS DYNAMIC TAINT ANALYSIS

The primary limitation of all dynamic taint analysis implementations is the runtime performance penalty imposed upon the guest system under analysis. This penalty becomes even greater when multiple taint sources are tracked separately using unique taint labels. Tracking the propagation of multiple taint labels requires either a single heavyweight taint propagation operation that accommodates all tracked labels or multiple lightweight taint propagation operations (one for each tracked label). Neither of these approaches scale when using a large number of taint labels, imposing a limit on the number of taint labels in use simultaneously.

DECAF ameliorates this limitation by performing precise, lightweight taint *status* propagation inline with guest execution while an asynchronous, heavyweight taint propagation of multiple taint *labels* is performed in parallel to the guest execution. DECAF implements its lightweight taint propagation mostly at the TCG instruction level, so it is easily extended to support multiple CPU architectures. To achieve bit-level precision, DECAF propagates tainted bits through CPU registers, memory and IO devices.

5.1 Taint Propagation in CPU Registers

DECAF creates TCG global variables to shadow the TCG global variables which represent general-purpose and flag CPU registers. Each shadow variable is the same size as the variable that it shadows, and each bit of the shadow variable represents the taint associated with the analogous bit in the variable. For example, the global variable `eax` for an x86 guest is shadowed by `taint_eax`, `ebx` is shadowed by `taint_ebx`, etc. When `eax` contains tainted data, `taint_eax` contains a bitmask that marks which bits of `eax` are tainted. These shadow variables emulate a set of dedicated taint-tracking registers in the guest CPU. DECAF also creates a shadow temporary variable on-the-fly to shadow each temporary variable present inside each TB. For the x86 target, we create shadow variables for the `cc_src`, `cc_dst` global variables so that taint propagates to CC flags naturally.

Once TCG translates guest instructions into a TB containing TCG instructions, DECAF performs a translation pass on the TB to insert additional TCG instructions which implement taint propagation rules that shadow each of the original TCG instructions. For example, Figure 5 shows that the instruction `mov_i32 tmp11, eax` is shadowed by `mov_i32 tmp21, taint_eax`. Some tainting rules are far more complex in order to be precise. For example, the `add` operation in Figure 5 requires nine extra TCG instructions to precisely propagate the taint bits from two source operands to the destination. DECAF's tainting rules have been formally verified to be sound (guarantee of no under-tainting at instruction level), and most of them have also been verified to be precise (guarantee of no over-tainting). The details are documented in our technical report [23].

Figure 5 illustrates this instrumentation pass. TCG translates a basic block of guest instructions into a TB of TCG instructions (a). DECAF performs its instrumentation pass on this TB by first performing a variable liveness analysis on the TCG code to determine if any TCG instruction is unnecessary or redundant. A TCG instruction that fails this analysis will be removed by TCG's optimization later, so there is no need to instrument it. Each opcode to be instrumented is compared against DECAF's list of tainting rules to determine which TCG instructions must be inserted to instrument it. The instrumentation TCG instructions are inserted prior to the original TCG instruction because some tainting rules (e.g. `and`, `or`) depend upon the values held in both the variables and shadow variables when determining taint propagation. Values held in the variables may change if the same variable is used as both the source and destination of the TCG instruction. Once this pass is complete, the TB now contains both the original and instrumentation code (b). The TCG engine performs an optimization pass on the instrumented TB and generates the final, optimized TB (c), which is then translated into the native instructions of the host and executed.

5.2 Taint Propagation in Memory and IO Devices

The guest's physical RAM is shadowed bit-for-bit by a three-level shadow page table. While other instrumentation platforms perform byte-level precision tainting of RAM[20, 24, 26] by representing each byte of taint as a single bit, that approach requires bit masking and shifting operations to represent a 32-bit register in a 4-bit space. DECAF's bit-level precision of shadow memory ensures that taint precision is not lost as taint propagates throughout the guest.

An implementation challenge is to re-factor the existing TCG instructions that access guest memory (`qemu_ld/st`) to also access shadow memory at the same time. This is nec-

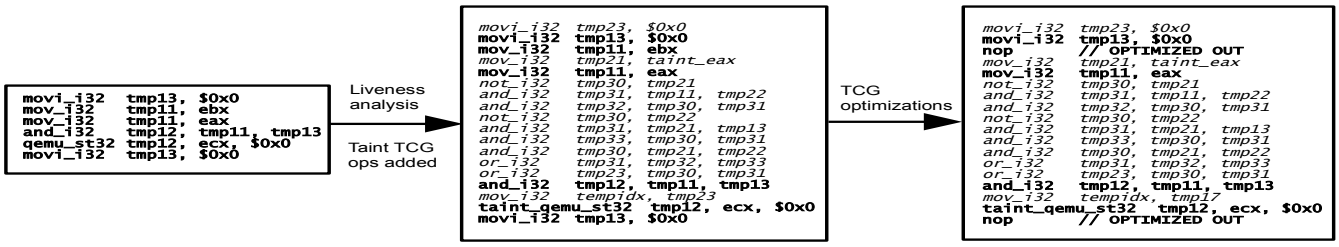

Figure 5: Register liveness tests determine which TCG instructions in the TB (a) should be instrumented for taint propagation, and instrumentation is inserted as needed (b). TCG's optimization logic eliminates unnecessary opcodes, resulting in an optimized, instrumented TB (c).

essary to ensure that taint propagation occurs at the same time that memory accesses occur. The inlined SoftMMU code already uses most of the host's x86 registers for TLB lookup and parameter passing, meaning that the stack must be used for passing taint information. This causes performance degradation and potential side effects if unexpected register spillage occurs when taint information is fetched from the stack. To counter this problem, additional shadow global variables are used specifically for copying taint information to and from the shadow page table.

Taint propagation in DECAF's virtualized devices (NE2000 NIC, IDE hard disk, PS/2 keyboard) is similar to taint propagation in memory. Each instrumented virtual device has a device-specific shadow memory, and a specific global variable passes taint data back and forth between device and RAM when programmable I/O or DMA operations occur.

5.3 Asynchronous Tainting

DECAF's lightweight taint propagation occurs inline with guest execution so that DECAF can halt execution at the exact moment that taint reaches a specific taint sink (i.e., instruction pointer, system call, virtual device). Asynchronous heavyweight taint propagation relies upon DECAF's Instruction Tracer plugin to efficiently log the taint propagation history. While the plugin is designed to log TCG instructions to record instruction traces, DECAF's flexible plugin interface enables Instruction Tracer to also record memory accesses, CPU states, and taint events. The plugin quickly logs enough information about the taint propagation for the log to be processed asynchronously by any custom analysis tool that executes as a separate process. Such tools can consume the taint log information as it is generated (running simultaneously with DECAF) or after DECAF's taint log has completed, performing a much more heavyweight taint analysis on the trace (i.e. reconstructing taint labels and propagation via backward slicing). The combination of lightweight and heavyweight taint tracking guarantees that taint detection is both timely and more scalable than the inline tracking of multiple taint labels.

Figure 6 shows the steps of the logging process. As each TB begins execution, the plugin writes an identifier for the TB and the current taint state of the CPU registers (a) to a staging buffer (b). If the TB has not been logged previously, or the TB has been flushed and retranslated since it was last logged, all TCG instructions and their arguments held in the TB are written to the staging buffer. Only the original, non-instrumented TCG instructions are written. Any memory and shadow memory accesses (both access size and both the virtual and TLB-resolved physical addresses) are written, as are the introduction of any new taint labels. As each group of TCG instructions implementing a single guest instruction

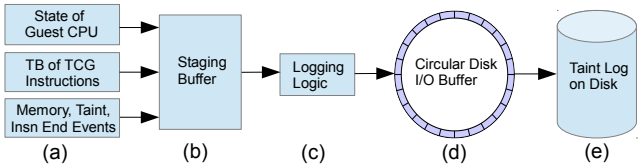

Figure 6: All events(a) are logged into a staging buffer(b). Logging logic(c) decides which events should be recorded and places them into a circular buffer(d) that is asynchronously written to disk(e).

complete execution, an "instruction end" event is recorded in buffer. This is necessary because TB execution can cease early due to jumps, branches, and exceptions. There must be a record of what instructions in the TB were executed so that execution can be reconstructed. When the execution of the next TB begins, the staging buffer is examined (c). If any global shadow variable contains taint, shadow memory is accessed, or a shadow memory location is marked with a taint label, the buffer is written to a circular buffer (d) that asynchronously writes log data to disk (e). Otherwise, the staging buffer is discarded.

6. EVALUATION

We evaluated DECAF with respect to the performance overhead under different configurations (such as VMI and tainting), and the analysis capabilities using three plugins (API Tracer, Keylogger Detector, and Instruction Tracer). An artifact has been accepted as part of this paper that can be used to partially reproduce our evaluation experiments. The artifact and source code for the plugins are available for download from DECAF's project page[7].

The hardware used for all evaluations is a 32-core 2.0GHz Intel Xeon ES-2650 CPU server with 128 GB of RAM. The server uses Ubuntu 12.04 Linux (3.2.0 kernel) as its OS. DECAF was executed on this server using an ARM Debian 6.0 Linux (2.6.32 kernel) VM image and three x86 guest VM images: Windows 7, Windows XP SP3 and Ubuntu 12.04 Linux (3.2.0 kernel). 4 GB of RAM was allocated to each of the x86 VMs, and 128 MB of RAM was allocated to the ARM VM. The priority of DECAF was nice'd to -20 to ensure it would be minimally influenced by other processes executing on the benchmark hardware.

6.1 SPEC CPU2006 Benchmarks

We evaluated its performance impact using the CINT2006 integer component of the SPEC CPU2006 benchmark suite.[2]

[2]462.libquantum was omitted from the test suite due to Visual Studio's Visual C++ not supporting some C++ features used by the test.

Table 1: Execution Overhead for DECAF and DECAF with VMI on different architecture/OSs without tainting.

Setup	XUbuntu	WinXP SP 3	Debian Squeeze (ARM)
DECAF with VMI	3m 25.9s	1m 4.36s	2m 50.16s
QEMU 1.0.1	2m 45.85s	0m 52.79s	2m 36.52s
DECAF + VMI Overhead %	24.14	21.91	8.72

Table 2: Code breakup of DECAF, VMI and different plugins. The code introduced by DECAF is beyond QEMU, which by itself has over 500K LOC.

	OS/Arch independent (LOC)	OS Specific (LOC)	Total (LOC)
DECAF	18470	1350	19820
Insn Tracer	3770	90	3860
API Tracer	840	880	1720
Key Logger	120	0	120

We chose the CINT2006 tests because the tainting instrumentation is applied to the TCG instructions, which all implement RISC-like integer operations. Floating point operations are implemented as a set of guest architecture-specific helper functions. Performance of ARM VMs under DECAF cannot be measured using the benchmark suite due to the memory requirements of the tests. The majority of the tests exceed RAM allocated to the VM.[3] and will measure the performance of the memory paging to disk, rather than the instrumented operations of interest. While a direct comparison of TEMU and DECAF performance using these benchmarks would be informative, it is infeasible because TEMU is too slow to correctly execute the tests. We attempted to execute the benchmark suite under TEMU, but the first benchmark test of the suite (400.perlbench) was allowed to run for over a day before its execution was terminated.

Baseline DECAF without any instrumentation experiences an average of 15.20% overhead over the execution performance of a similarly-configured QEMU. DECAF updates `EIP` (x86) and `R15` (ARM) after every guest instruction to ensure accurate analysis, while QEMU updates these registers at the end of each TB. DECAF must also maintain its plugin infrastructure by continually watching for the registration of new plugin callbacks.

The VMI overhead measurements in Figure 7 show the difference in performance between running DECAF in a baseline configuration with all features disabled and a configuration with only VMI enabled. Average overhead is 12.07% for Windows 7 and 14.48% for Linux. The negative overhead result for the Linux 400.perlbench test can be attributed to the short execution time of the test and the general variability in execution times within an emulated VM environment. The result of 429.mcf has considerably higher VMI overhead than the other tests with 54.36% for Windows 7 and 55.23% for Linux. This test incurs almost twice as many TLB misses as the next closest test (471.omnetpp). VMI callbacks are triggered when TLB misses occur, explaining the larger amount of observed overhead.

Furthermore, Table 1 and Table 2 present the boot time overhead and source code distribution between architecture dependent and independent components. DECAF and VMI impose a combined overhead under 25% on x86 and 8.72% on ARM. Also, from Table 2 we can see that most of the code in the plugins DECAF are architecture independent. API Tracer includes OS specific code to interpret some OS specific data structures however, the core part of API Tracer contains no OS specific code.

[3]The "versatilepb" platform QEMU uses to emulate ARM VMs has a 256MB RAM limitation.

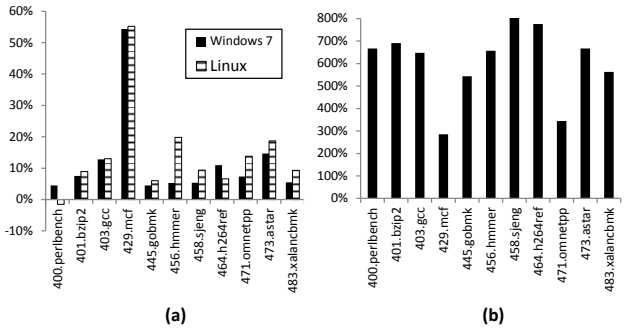

Figure 7: CINT2006 benchmarks that measure overhead for VMI (a) and inline taint propagation (b).

The inline taint propagation measurements in Figure 7 show the difference between running DECAF in a baseline configuration with all features disabled and with inline tainting enabled for the Windows 7 VM.

The average overhead is 605.07%, ranging from 285.32% (429.mcf) to 815.77% (458.sjeng). Taint propagation overhead is directly related to the number of TCG instructions being executed, so it is highest for CPU-bound tests. Because DECAF's inline tainting executes multiple taint propagation TCG instructions for each TCG instruction that executes within the whole system, an average slowdown of six-times is justified.

6.2 API Tracer

The API Tracer leverages the VMI and function hooking features of DECAF to capture API-level traces pertaining to user- and kernel-mode execution of a program.

At its core, API Tracer is a minimal and stand-alone cross-platform component comprising 340 lines of C code that retrieves function-level execution traces of programs on any platform/OS supported by DECAF. Furthermore, we implement a custom configuration parser comprising 500 lines of C code and a Windows-specific extension component comprising 880 lines of C code to decipher the higher-level OS-specific semantics. For example, in Windows the `kernel32.dll::CreateProcess()` API call contains newly created process information and the creation flag parameters required to extend analysis into child processes. The OS-specific component interprets such information and acts accordingly. Unlike static-analysis based tools that can not analyze dynamically generated code, and user-space dynamic analysis tools (such as Pin [15]) that can not analyze activity in the kernel, API Tracer keep tracks any kernel modules loaded by a user program and traces such modules automatically. It also monitors the memory allocation and deallocation of a program to identify and trace unpacked/dynamically generated code, thereby providing rich cross-platform and system-wide analysis capabilities.

Figure 8 shows the overhead introduced by API Tracer on Windows XP SP3 as it scales with the number of functions in the plugin's configuration file [4]. DECAF selectively

[4]Configuration file consists of all the methods that need to be captured along with the parameter list/types, return type and calling convention.

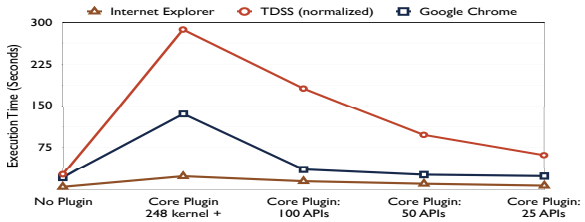

Figure 8: Evaluation of API Tracer plugin.

instruments only the TBs that correspond to the hooked functions, thereby significantly improving performance. An un-optimized implementation would be to instrument *all* basic blocks and filter the ones that correspond to hooks - similar to what TEMU [20] does. As a comparison, Internet Explorer loads the webpage (www.gnu.org) in 22.6 seconds and 217.79 seconds with selective optimization on and off respectively. For the sake of evaluation, we considered two popular web browser clients for Windows - IE and Chrome, and a notorious bot TDSS [11] that inserts a kernel module to hide itself in the kernel. API Tracer is not only able to trace the inserted kernel module, but is also able to extract the unpacked code in memory for further analysis. The Chrome browser uses a *multiple-processes architecture* and keeps tabs, extensions, web apps, and plug-in processes independent from each other and spawns new processes as and when required. API Tracer is able to automatically trace the parent Chrome process and the child processes.

6.3 Keylogger Detector

Keylogger Detector is an extended version of the sample plugin in Figure 2. Leveraging the VMI, tainting, and event-driven programing features of DECAF, this plugin is capable of identifying keyloggers and analyzing their stealthy behaviors. The core of Keylogger Detector is cross-platform and OS-independent, comprising only 120 lines of C code.

By sending tainted keystrokes into the guest system and observing if any untrusted code modules access the tainted data, we can detect keylogging behavior.The sample plugin can introduce tainted keystrokes into the guest system and identify which modules read the tainted keystroke by registering `DECAF_READ_TAINTMEM_CB` and `DECAF_KEYSTROKE_CB` callback events. To capture the detailed stealthy behaviors, Keylogger Detector implements a shadow call stack by registering the `DECAF_BLOCK_END` callback. Whenever the callback is triggered, we check the current instruction. If it is a `call` instruction, we retrieve the function information using VMI and push the current program counter onto the shadow call stack. If it is a `ret` instruction and pairs with the entry on the top of the shadow call stack, we pop it from the stack. When the `DECAF_READ_TAINTMEM_CB` callback is invoked, we retrieve information about which process, module, and function read the tainted keystroke data from the shadow call stack.

To evaluate our Keylogger Detector, we collected a set of malware samples that are known to have key-logging functionality. This sample set has 117 malware samples in total, spanning 29 malware families. We tested them on Windows XP SP3 by sending keystrokes to the `notepad` application and observing whether any tainted keystrokes were accessed by the tested sample. Keylogger Detector successfully detected the keylogging behaviors in all of these samples. Table 3 is the trace of Trojan.Win32KeyLogger. It shows which module of the process read the tainted keystroke using which function. From the trace, we can tell that the tainted keystroke enters the system and is fetched by the untrusted code of `MPK.exe`, which clearly depicts a keylogging activity. Furthermore, the trace shows which functions were used to steal keystrokes. This information is very valuable for performing malware analysis.

Table 3: Trojan.Win32.KeyLogger Trace.

PROCESS	MODULE	FUNCTION
<KERNEL>	i8042prt.sys	hall.dll:READ_PORT_UCHAR
<KERNEL>	win32k.sys	ntoskrnl.exe:PsGetProcessWin32Process
<KERNEL>	win32k.sys	hal.dll:HalEndSystemInterrupt
...	...	...
notepad.exe	Mpk.dll	ntoskrnl.exe:ProbeForWrite
notepad.exe	Mpk.dll	user32.dll:SendMessageA
...	...	...
MPK.exe	user32.dll	ntoskrnl.exe:ProbeForWrite
...	...	...
MPK.exe	MPK.exe	kernel32.dll:InterlockedIncrement
...	...	...
MPK.exe	MPK.exe	hal.dll:HalEndSystemInterrupt
MPK.exe	MPK.exe	ntdll.dll:wcscpy
MPK.exe	user32.dll	ntoskrnl.exe:ProbeForWrite
...	...	...

6.4 Instruction Tracer

Leveraging VMI, tainting, and instruction execution callbacks provided by DECAF, Instruction Tracer records a TCG instruction-level trace with concrete and taint values for a specific user-level process or kernel code region. Similar to the other two plugins, Instruction Tracer is largely platform-neutral, capable of collecting execution traces for programs in x86 and ARM, Linux and Windows. Moreover, it is also easier to perform formal verification on the TCG trace, due to its RISC-like instruction semantics. For example, it has been demonstrated to be feasible to convert the TCG trace into LLVM IR and perform symbolic execution on the trace [3]. Instruction Tracer is implemented in 3860 lines of C code, though this includes the code for both the plugin and the parser for the log file that the plugin generates.

To demonstrate the practical effectiveness of this plugin, we used Instruction Tracer to detect a buffer overflow at runtime. The sample code in Figure 9 was compiled and executed inside of x86 and ARM Linux VMs running under DECAF with Instruction Tracer loaded.

```
1. int func1(char *input) {
2.   char buffer[4];
3.   strcpy(buffer, input);
4. }

5. void main(void) {
6.   char buffer[16];
7.   scanf("%s", buffer);
8.   func1(buffer);
9. }
```

Figure 9: A simple buffer overflow example.

The code contains a simple buffer overflow vulnerability. If more than three characters are entered by the user, `buffer` in `func1()` will overflow and begin corrupting data stored on the stack. To capture the corruption, characters are entered into the program via tainted keypresses until the return address is modified by the overflow. Under the ARM environment, Instruction Tracer identified the buffer overflow when `R15` (PC) became tainted after entering five characters. `R14` (Link Register) was also monitored for taint, but it never became tainted during the test. Figure 10 shows the log output at the point where `R15` first becomes tainted. Tainted character data is fetched from stack memory, masked to ensure that the value is properly aligned, and then stored in `R15`.

Under the x86 environment, the global variable for the `EIP` register can't be directly passed to an opcode as an argument. `EIP` is modified by writing to host memory via the `st_i32` opcode. Watching for tainted writes to `EIP`'s offset

```
qemu_ld32 tmp61[00000000],tmp50[00000000],$0x0
 --> TAINT HAS BEEN READ FROM MEMORY:
     Address: 0x07837e5c (4 bytes)
     Taint: [ffffffff]
movi_i32  tmp62[00000000],$0xfffffffe
and_i32   pc[00000000],tmp61[00000000],tmp62[ffffffff]
 --> TAINT NOW PRESENT IN PROGRAM COUNTER (R15)
```

Figure 10: Buffer overflow detection on ARM.

(0x4C) in the `CPUState` data structure identifies that the buffer overflow. Figure 11 shows the log output at the point where `EIP` first becomes tainted. Tainted character data is fetched from memory located at the address in `ESP`, the stack size is reduced by four bytes, and the tainted data is then placed into `EIP`'s offset in the `CPUState` data structure.

```
mov_i32   tmp2[00000000],esp[00000000]
qemu_ld32 tmp0[00000000],tmp2[00000000],$0x0
 --> TAINT HAS BEEN READ FROM MEMORY:
     Address: 0x0bfffff30 (4 bytes)
     Taint: [ffffffff]
movi_i32  tmp15[00000000],$0x4
add_i32   tmp4[00000000],esp[00000000],tmp15[00000000]
mov_i32   esp[00000000],tmp4[00000000]
st_i32    tmp0[ffffffff],env,$0x4c
 --> TAINT NOW PRESENT IN EIP
```

Figure 11: Buffer overflow detection on x86.

We also performed a comparison of Instruction Tracer's performance against that of the TEMU's Tracecap plugin. Tracecap generates a trace of the guest's instructions as they execute to facilitate analyses similar to that of the buffer overflow analysis performed with Instruction Tracer. We used DECAF and TEMU to emulate the same Windows XP VM and trace the execution of an instance of the DOS `sort` application. For both plugins, tainting was disabled. A text file 5.4 MB in size was selected to be sorted, and both plugins were configured to log their execution traces of the application directly to `/dev/null`. The sort completed in 39m 57.33s with Tracecap running, but in only 2m 5.23s with Instruction Tracer running (almost 20 times faster). The same sort with a stock QEMU completed in 5.89s.

7. RELATED WORK

Several instrumentation solutions perform data flow analyses within the scope of a single process or binary. Such solutions are generally much faster than whole-system analysis platforms like DECAF and operate directly upon the native instructions of the binary under analysis. The Pin[15] API is a flexible C/C++ interface that is used to create instrumentation tools (known as "Pintools"). Examples of such Pintools are libdft[14] and DYTAN[5]. Pintools do not have the benefit of having a plugin development API that works at a semantic level higher than individual instructions, like DECAF does. DYTAN is designed as a platform for prototyping different tainting policies, while DECAF relies upon a proven sound and precise policy. libdft offers a less flexible, but faster, solution for tracking explicit data flows, but it is has the same limitations of other Pintools and, unlike DECAF, only supports instrumenting x86 binaries.

Many efforts have been made to reduce the runtime overhead of dynamic taint analysis. LIFT[19] assumes that taint propagation is not needed for most code execution, so it optimizes performance by taking the "fast paths" (without taint instrumentation) most of time. It also exploits extra registers in 64-bit architectures to shadow taints in 32-bit applications. Minemu[2] leverages the x86 SSE registers to provide lightweight taint tracking for 32-bit x86 applications. Jee et al[12] build upon libdft to create a system that performs a static analysis on a process to selectively instrument the process for dynamic analysis per the rules of a Taint Flow Algebra. All these tainting implementations only track taint status, and apply imprecise and sometimes unsound tainting rules, to achieve high efficiency. In comparison, DECAF is designed to perform accurate binary analysis in offline settings. So we cannot sacrifice precision and correctness for efficiency. DECAF is also designed to be generic, so we avoid relying on architecture-specific features (e.g., SSE) to boost up performance. Using static analysis to guide selective taint instrumentation is appealing, but is not generic and scalable in the whole-system setting.

Whole system instrumentation platforms leverage binary emulation and VMI. Early systems, such as TaintBochs[4], favor accuracy over performance. Ether[8] attempts to elude and analyze VM-aware malware by leveraging Intel VT hardware virtualization extensions. By triggering a debug exception after every instruction, Ether is able to fully analyze the state of the system, at the cost of heavy execution overhead.

DECAF seeks to perform practical, accurate analyses of interactive systems, making the reduction of such high overhead a focus of its design. Like DECAF, TEMU[21] is built upon QEMU. It serves as the base for a variety of security analysis tools that perform whole-system analysis, such as HookFinder[25], Panorama[26], and Renovo[13]. TEMU is based upon version 0.9.1 of QEMU, which uses the older, defunct "dyngen" system (rather than TCG) for binary translation. TEMU is also not capable of emulating newer OSes such as Windows 7 and 8, and it is only capable of instrumenting x86 platforms. DECAF is capable of emulating these OSes and the ARM platform. S2E[3] uses QEMU to perform inline symbolic execution. When execution of the guest environment reaches a branch within code of interest, S2E forks the current QEMU process to explore both branches using symbolic execution. While powerful, this process is quite slow and memory intensive. DECAF is designed to assist in performing such heavyweight analyses by using lightweight plugins to capture detailed system information and instruction traces that provide enough detail to allow other tools to perform heavyweight analyses offline.

DroidScope[24] is a dynamic analysis platform for security analysis on Android. The core idea of DroidScope is to seamlessly reconstruct both Dalvik-level and OS-level semantic views and to provide a unified interface for Android malware analysis. DroidScope is an extension to DECAF for Android-specific analyses.

8. CONCLUSIONS

We present DECAF, a QEMU-based, multi-target, whole-system dynamic binary analysis framework. It implements a novel method of VMI and explicit data flow tracking that incur much smaller runtime performance penalties than those seen in other whole-system analysis platforms. It provides a simple, event-driven plugin API for the development of largely platform-neutral analysis software.

9. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their comments. This research was supported in part by NSF Grant #1018217, NSF Grant #1054605, and McAfee Inc. Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

10. REFERENCES

[1] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[2] E. Bosman, A. Slowinska, and H. Bos. Minemu: The World's Fastest Taint Tracker. In *Recent Advances in Intrustion Detection*. Springer, 2011.

[3] V. Chipounov, V. Kuznetsov, and G. Candea. s2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[4] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (Security'03)*, 2004.

[5] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA'07)*, 2007.

[6] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*, 2004.

[7] DECAF Binary Analysis Platform - "Taking the jitters out of dynamic binary analysis". `https://code.google.com/p/decaf-platform/`.

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

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2011.

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

[11] S. Golovanov. Analysis of tdss rootkit technologies. Technical report, Securelist, 2010.

[12] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2012.

[13] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)*, 2007.

[14] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments - VEE '12*, 2012.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[16] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[17] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[18] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys 2006*, 2006.

[19] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, 2008.

[21] TEMU: The BitBlaze dynamic analysis component. `http://bitblaze.cs.berkeley.edu/temu.html`.

[22] X. J. X. Wang and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of ACM Conference on Computer and Communication Security*, 2007.

[23] L. K. Yan, A. Henderson, X. Hu, H. Yin, and S. McCamant. On soundness and precision of dynamic taint analysis. Technical Report SYR-EECS-2014-04, Syracuse University, 2014.

[24] L. K. Yan and H. Yin. DroidScope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[25] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behavior. In *15th Annual Network and Distributed System Security Symposium*, 2008.

[26] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of ACM Conference on Computer and Communication Security*, 2007.