# On-Chip System Call Tracing: A Feasibility Study and Open Prototype

Chengyu Zheng*, Mila Dalla Preda†, Jorge Granjal‡, Stefano Zanero* and Federico Maggi*
* DEIB, Politecnico di Milano, Italy
Email: {name.surname}@polimi.it
† Dipartimento di Informatica, University of Verona, Italy
Email: mila.dallapreda@univr.it
‡ CISUC, University of Coimbra, Portugal
Email: jgranjal@dei.uc.pt

***Abstract*—Several tools for program tracing and introspection exist. These tools can be used to analyze potentially malicious or untrusted programs. In this setting, it is important to prevent that the target program determines whether it is being traced or not. This is typically achieved by minimizing the code of the introspection routines and any artifact or side-effect that the program can leverage. Indeed, the most recent approaches consist of lightly instrumented operating systems or thin hypervisors running directly on bare metal.**

**Following this research trend, we investigate the feasibility of transparently tracing a Linux/ARM program without modifying the software stack, while keeping the analysis cost and flexibility compatible with state of the art emulation- or bare-metal-based approaches. As for the typical program tracing task, our goal is to reconstruct the stream of system call invocations along with the respective un-marshalled arguments.**

**We propose to leverage the availability of on-chip debugging interfaces of modern ARM systems, which are accessible via JTAG. More precisely, we developed OpenST, an open-source prototype tracer that allowed us to analyze the performance overhead and to assess the transparency with respect to evasive, real-world malicious programs. OpenST has two tracing modes: In-kernel dynamic tracing and external tracing. The in-kernel dynamic tracing mode uses the JTAG interface to "hot-patch" the system calls at runtime, injecting introspection code. This mode is more transparent than emulator based approaches, but assumes that the traced program does not have access to the kernel memory—where the introspection code is loaded. The external tracing mode removes this assumption by using the JTAG interface to manage hardware breakpoints.**

**Our tests show that OpenST's greater transparency comes at the price of a steep performance penalty. However, with a cost model, we show that OpenST scales better than the state of the art, bare-metal-based approach, while remaining equally stealthy to evasive malware.**

## I. Introduction

With over 500 million devices and an estimated 84% market share [1], Android-based devices are the cyber-criminals' main target in the mobile world. In addition to the alarming amount of malware families and samples [2], the evasive techniques employed by malware are becoming more and more sophisticated. With the high amount of new applications being released every month, researchers and app-store administrators are striving to find reliable solutions to analyze apps in order to recognize and isolate malicious ones.

This work has been partially supported by MIUR FACE Project No. RBFR13AJFT

**Research Gap.** Static analysis approaches [3] allow us to obtain sound information on entire portions of an application. However, Code obfuscation and dynamic code-loading techniques are often used to obstruct static analysis—for both benign and malicious purposes. The idea of these techniques is to make the results of static analysis tools so imprecise to become unusable in practice. As a result, the research community has been very active in the area of dynamic analysis, with the goal of reconstructing a precise picture of the events occurring during program execution (e.g., operating system procedures invoked, network-level events, content of memory).In dynamic analysis approaches there is a well-known trade off between transparency and semantic richness. On the one hand, typical dynamic approaches, such as TraceDroid [4], instrument the runtime or the virtual machine (i.e., Dalvik machine, in the case of Android) and are able to capture high-level events (e.g., API function calls) and data structures (e.g., Java types). Higher-level behaviors, such as "sending a spam email" or "opening a reverse shell on port X", are then reconstructed by using system calls as "basic blocks". This type of introspection is prone to evasion, due to the use of instrumentation artifacts that a malicious program can leverage to detect whether or not it is being traced. On the other hand, attempting to mitigate this problem, researchers have proposed virtual-machine introspection (VMI) techniques. For instance, CopperDroid [5] and DroidScope [6] follow this approach and move the introspection scope below the operating system, in a full-system emulator. VMI-based approaches track the occurrence of a system call by assuming that the traced program uses the calling convention according to the operating system's application binary interface(ABI). This allows them to reconstruct high-level events, as if the operating system was instrumented directly, while keeping their visible footprint low. Unfortunately, VMI-based approaches are not resilient to evasion [7], [8]. For example, in [9] the authors show how emulator-detection routines can be generated automatically. For Windows/x86 platforms, the problem has been mitigated by moving the dynamic analysis of malicious programs to the bare metal [10], [11]. In these approaches, the introspection task is performed in a thin hypervisor, or slightly modified operating system. MALT [12] follows this approach by leveraging hardware support and BIOS modifications to further reduce the introspection code base's footprint. Recently, we observe the same trend for the Linux/ARM platform. BareDroid [13] elegantly leverage the SELinux layer to trap the calls to operating-system procedures. In this way, BareDroid effectively obtains a lightweight introspection system on unmodified Android images running on real devices rather than on emulators. However, in this solution, SELinux (i.e., the kernel) is still assumed to be outside the attacker's model.

**Proposed Approach.** Following the aforementioned trend, we investigate the feasibility of moving the introspection code for Android applications completely outside the analysis system. In other words, we investigate the feasibility of porting the current VMI-based approach in a hardware system, rather than on an emulator. The main challenge is to be able to efficiently introspect the machine state in order to observe events such as instructions, interrupts, and registers, that are needed for reconstructing the occurrence of a system call. Our idea is to leverages the fact that ARM-based systems are more open than x86-based systems, and thus offer built-in tracing functionality. Indeed, on-chip debuggers for ARM are available in off-the-shelf development boards at affordable prices and accessible through standard JTAG interfaces.

In this work, we focus specifically on assessing the overhead imposed by the tracing process, the resilience of the proposed tracing approach to evasion techniques, and the technical feasibility of the overall idea. To this end, we propose a set of open-source tools and a working prototype, that we name OpenST, to run an executable Linux/ARM application and trace its system calls together with their arguments. OpenST can trace system calls by running introspection routines in two modes: external tracing and in-kernel dynamic tracing. The external tracing mode consists in executing the introspection code on an external computer. Observe that this mode is conceptually equivalent to, but more stealthy than a VMI-based approach. The in-kernel dynamic tracing mode executes the introspection code on the target machine. For this reason, it is faster but less transparent than the external tracing mode. More specifically, in the external tracing mode, OpenST sets an hardware breakpoint whenever a software interrupt is trapped; the breakpoint handler pauses the CPU, inspects its registers and the target process' memory, and resumes once done with collecting the necessary data. As part of our approach, we generate the introspection code automatically, offline, by parsing the system-call prototypes from the kernel binary (in DWARF format). For each system-call definition we parse the data structures of its input arguments and convert them in offsets, which we use to instruct the stub code to unmarshall the argument values from the main memory given a base address. In the in-kernel tracing mode, the introspection code is injected into the target machine at runtime. More precisely, through an inline hooking mechanism of the SWI handler, OpenST is able to dynamically detour the execution to the introspection code, which will dump the system call and its arguments and transfer its content over UDP packed to a remote machine.

**Evaluation.** We implemented a testbed application that we specifically developed to precisely measure the overhead at the system call level. We performed a set of micro benchmarks on OpenST. As a side result, these tests showed that OpenST correctly intercepts the system calls and un-marshalls their arguments.

We then tested the resilience of OpenST against real-world malicious applications that are known to perform emulator evasion (i.e., the `Android.HeHe` and `Android.Pincher.A` malware families). When compared to a SELinux-based tracing approach running in an emulator, OpenST was able to observe two order of magnitude more system calls, confirming its resiliency to emulator-based evasion techniques.

Finally we performed a macro benchmark to measure the performance of OpenST. Our results show an overhead for the external tracing mode, which in the worst cases can reach a $74\times$ slowdown with respect to an emulator-based approach. The in-kernel tracing mode, while less transparent than the external mode, has nearly the same execution time of an emulator-based approach, while being more evasion-resilient.

**Original Contributions.** In summary, in this paper we make the following contributions:

- We are the first to investigate the feasibility of performing precise process introspection using on-chip debugging interfaces and low-cost equipment;
- We design, implement and release a working, open prototype that other researchers and practitioners can use to trace untrusted Linux/ARM programs;

In the spirit of open science, the source code of OpenST is available at https://github.com/necst/openst.

## II. Background and Motivation

In this section we describe the limitations in the state of the art of introspection techniques, their limitations (which motivated our work) and our main goals.

### A. *Shortcomings in dynamic analysis*

Dynamic analysis techniques are widely used to extract the runtime behavior of programs. Dynamic analysis tools typically use some form of instrumentation in order to intercept the events of interest during execution (e.g., machine instructions, kernel system calls, userland API functions). A common technique is to modify of the kernel such that to intercept the system calls. Recently, full-system emulators and virtual machines (e.g., QEMU, XEN) are prefered for dynamic analysis techniques, because they provide enough visibility and control of memory and machine state (up to the granularity of the single instruction), while ensuring great transparency, sitting outside the monitored system. However, it is well known that emulators and virtual machines do not precisely reproduce the dynamics of real systems [14]. Indeed, most of the times, there are small discrepancies in how instructions are implemented in real vs emulated systems. These discrepancies can be leveraged by the monitored program in order to determine whether it is running in a real or emulated system. Therefore, if the monitored program realizes that it is executing in an emulated system, it can refuse to exhibit its true (malicious) behavior. In complex emulated systems, such as modern phones, endowed with GPS and accelerometers, these discrepancies can be striking. For example, in [7], [8] the authors show that the Android emulator is vulnerable to evasion techniques based on hardcoded IMEI, GPS coordinates, accelerometer activity, or caching behavior. Moreover, evasion techniques can leverage instrumentation code. For example, unavoidably leave artifacts in memory or in the runtime behavior of the monitored system, which can be exploited by the malicious program in order to compromise the correctness and completeness of the analysis.

### *B. System Call Introspection in Linux/ARM*

CopperDroid [5] and DroidScope[6] are sandboxes based Android emulator. In these tools the "instrumentation" of the emulator is performed at the virtual CPU level. More precisely they modify the code that the emulator executes when an exception or an interrupt occurs. In particular, they insert introspection code that is able to analyze memory content and dump it. This technique is called virtual machine introspection (VMI). For example, with VMI it is possible to use the emulator to inspect the memory and reconstruct the PID, the task group identification (TGID), the executable name (COMM), or the system call arguments. Through VMI, CopperDroid and DroidScope are able to reconstruct both OS- and Android-level semantics. User-space applications interact with the operating system thought system calls, according to the calling convention defined by the Application Binary Interface (ABI). For example, when a system call takes a non primitive type as argument, such as a struct, the pointer to the structure is stored into a register according to the calling convention. By introspecting the memory at the address given by the CPU register, one can find the struct. By assuming knowledge of the OS (e.g., structure definition or other type information), the raw memory content can be un-marshalled, so deriving the original, structured data.

### *C. Goal and Approach*

VMI techniques are useful to analyze an application whose source code is not available. However, when this technique is applied to emulator, the presence of an instrumented emulator can be easily spotted as discussed in Section II-A. Therefore, we want to investigate the feasibility of porting VMI techniques in hardware, and study if such an approach can help analyze applications that are known to implement anti-emulation techniques. We want to use VMI techniques with hardware support, without requiring any kind of cooperation from the operating system. For example, rebuilding the kernel would immediately reveal to the malware that it is being executed in a compromised non-real system. Currently, the only architecture that has hardware debugging facilities that offer the same visibility of an emulator are ARM processors.

Our approach is to rely on hardware debugging interfaces present in development boards, which allows us to communicate with ARM-based CPUs and to intercept interesting events such as system call invocations. This approach works conceptually on any CPU, but we focus our prototype on ARM processors due to the presence of JTAG interface. For the benefit of the research community, we designed OpenST to have an open design and implementation. Moreover, OpenST has low deployment costs that allow it to scale by adding more hardware as demanded by the throughput requirements. Our prototype is one order of magnitude less expensive compared to proprietary closed-source solutions, which hardware, consisting of a board and a tracer, would cost \$9,500 (\$6,000 Juno VExpress Board and \$3,500 of ARM DSTREAM Tracer) and its relative software license with price tag of \$6,000. Thus, with OpenST we aim at providing a cost efficient open-source system for the analysis of malware that employs evasive techniques.

Our OpenST prototype is implemented on a development board, but it can be adapted to other ARM-based devices with minimal effort. Clearly, in order to use it on an arbitrary phone or tablet, the JTAG pinout must be known. A typical use of OpenST would be in the analysis of applications for which we suspect the presence of evasive techniques.

## III. System Implementation

We implemented OpenST with two degrees of transparency: *In-kernel dynamic tracing* and *external tracing* mode. The in-kernel dynamic tracing mode uses the JTAG interface to "hot-patch" the system calls' at runtime, in the kernel code region.

Both the In-kernel dynamic tracing and external tracing mode divide the tracing process into three phases as illustrated in Figure 1.

### *A. External Tracing Mode*

In this mode, we set the breakpoints to intercept system calls, then we perform a filtering on the process identification (PID) or name (COMM) of the caller application, only if it match with the one under analysis we inspect the memory in order to reconstruct the system calls and its arguments.

*1) Phase 1: Hardware Breakpoint Management:* In this phase, we want to instrument the target board to intercept all the system calls. For this reason, we set a breakpoint to the SWI exception vector. The SWI exception vector in Linux (on which Android is based) is located eight byte after the beginning of the vector table which, in 32-bit architecture, is loaded into "0xffff0000" at boot time, thus we set the breakpoint at address "0xffff0008". Since halting the processor for every system call of every process is expensive, we propose an optimization that uses two breakpoints (hybrid breakpoint) instead of a single breakpoint. The goal is to halt the processor when the program counter (PC) matches the desired address and the context ID matches the program under analysis. Observe that the context ID is revealed by inspecting the memory when the first system call of the target application occurs. Fortunately, ARM-based processors allow the use of hybrid breakpoints as a composition of two Breakpoint Register Pairs (BRPs), where a BRP is composed by a Breakpoint Control Register (BCR) and a Breakpoint Value Register (BVR). Our idea is to store in the first BCR the address location of the

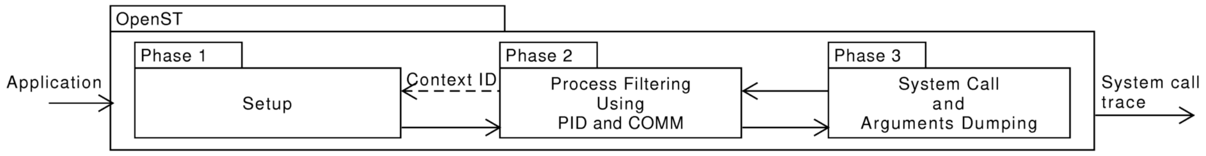


Fig. 1: OpenST logical architecture. Phase 1 instrument the target board. Phase 2 filter the system call based on the PID or COMM of the caller program. Phase 3 introspect the system call and dumps the unmarshalled data.

Listing 1: thread_info structure is used to extract the offset of task_struct

```
struct thread_info {
  long unsigned int   flags;
  int                 preempt_count;
  mm_segment_t        addr_limit;
  struct task_struct *task;  // offset: 0x00c
  ...
}
```

Listing 2: task_struct structure is used to extract the offset of PID and COMM.

```
struct task_struct {
  volatile long int   state;
  ...
  pid_t            pid;      // offset: 0x108
  ...
  char             comm[16]; // offset: 0x2b4
  ...
}
```

SWI exception vector, and in the second BCR the Context ID of the process. By linking these two breakpoints with the BVR we are able to trace only the system calls of the application under analysis [15], thus containing the overhead.

*2) Phase 2: Process Filtering:* In this phase we handle the callback procedure executed at each breakpoint. After hitting the breakpoint, the CPU is halted, thus allowing us to issue `read` and `write` of the system's memory in a consistent state. In fact, we are able to inspect the memory to obtain information about the current process.

In Linux/Android, process information is stored in the `task_struct` (see Listing 2) structure, which includes information such as PID and COMM. The address of this structure is stored in `thread_info`. The `thread_info` (see Listing 1) structure, which contains CPU-specific information, is stored at the end of the stack. Since the kernel stack is 8KB, and 8KB aligned, the location of `thread_info` can be retrieved by masking out the less significant bits of the stack pointer (SP) (e.g., `thread_info = sp && 0xfffc000`). Inside `thread_info` at offset `0xc` we find the pointer to `task_struct` structure. Then again, by reading at the appropriate offset of (as showed in Listing 2) `task_struct`, OpenST is finally able to retrieve the PID and COMM.

Whenever PID and COMM match the process under analysis, OpenST read the context ID register by using the `mrc` coprocessor instruction `arm->mrc(target, 15, 0, 1, 13, 0, &contextid)` [15]. At this point, the context ID is now available and forwarded to the first phase, with the use of hybrid breakpoint is possible to filtering to the Phase 1, thus increasing the performance.

*3) Phase 3: Memory Introspection & Argument Unmarshalling:* In this phase we introspect the memory to accurately reconstruct high-level information associated to the executed system calls and their arguments. ARM defines its own Embedded ABI (EABI). The EABI uses processor registers to pass system call arguments. The arguments are ordered left to right, starting from `r0` to `r6`, while the system call number is saved in the register `r7`. When the type of the passed argument is primitive it is possible to dump the system calls' argument directly by simply reading the content of the respective registers. When the type of the passed argument is not primitive, for example a structure, the registers stores only the pointer to the structure while the data of the structure is stored in memory. This means that in order to reconstruct arguments of non-primitive types we need to introspect both memory and registers.

To automate the process of creating the aforementioned introspection routines for each system call we developed a compiler that produces the introspection code based on the information extracted from the binary image (in DWARF format). Such semantic information is located in the DWARF section of the kernel binary, and it includes size, offset and type of the arguments. This information is used to issue read commands to the device at specified location and size in order to dump the data structure. In our implementation, functions that dump system calls are prefixed with `dump_sys_` while every function that dumps structures is prefixed with only `dump_`. For example `dump_sys_gettimeofday` recursively calls the `dump_` function related to their arguments, such as `dump_timeval` and `dump_timezone` (see Figure 2).

## *B. In-kernel Dynamic Tracing Mode*

This mode leverages introspection code to be injected directly into the kernel memory. The code is executed on the target board by using the JTAG to hijack the normal flow of execution of the SWI handler. In this way, we alter the memory of the target machine and therefore we sacrifice transparency, but we gain a substantial improvement in performance.

*1) Phase 1: Loading of the Introspection Code:* In this phase, we allocate an area of memory in the target machine and then load the introspection code. Therefore, OpenST uses

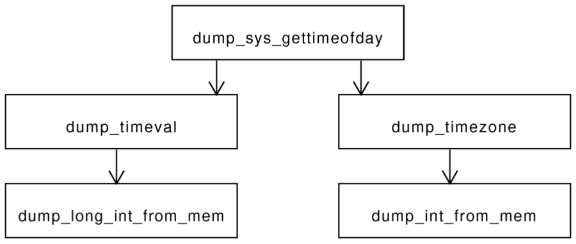


Fig. 2: An example of system calls introspection and the function being called in order to dump it.

Listing 3: This diagram contain a part of the introspection code.

```
+- _sys_entry ---------------------------+
| STMFD   SP!, {R0-R12,LR}               |
| BL      _disable_irq                   |
| BL      _disable_preemption            |
| BL      _get_pid_comm                  |
| LDR     R1, _pid                       |
| CMPNE   R0, R1                         |
| ...                                    |
| BE      _sys_entry_ret                 |
+----------------------------------------+
  | true                           |
  v                                |
+-------------------------------+  |
| ...                           |  |
| CMP     R7, #3                |  |
| LDMEQFD SP, {R0-R6}           |  |
| BLEQ    _log_sys_read_entry   |  |
| ...                           |  |
+-------------------------------+  |
  |                                | false
  v                                v
+- _sys_entry_ret -----------------------+
| BL      _enable_preemption             |
| BL      _enable_irq                    |
| LDMFD   SP!, {R0-R12,LR}               |
| B       _exit_entry                    |
+----------------------------------------+
```

vmalloc[1] to allocate a page of executable memory. In order to minimize the interference with other parts of the kernel, we set a breakpoint at the entry point of the vmalloc, we intercept it, and make it allocate two times. The first time with our parameters we allocate 4,096 byte of executable memory. The second time with its original parameters, so that, it can resume the normal execution flow. Next, we load the introspection code into the newly allocated memory. After that, we patch the entry point of the system call routine with a jump instruction to the introspection code by replaying the original one.

The introspection code that we inject is composed of a prologue, a main body, and an epilogue as showed in Listing 3. In the prologue we save the caller's registers and we lock the entire introspection code by disabling the interrupts and the preemption of the CPU, so interrupts are not nested and the CPU is not preempted with a context switch when the critical section is being executed. After that, OpenST call the _check_pid_comm routine, which returns true if either PID or COMM match the values of the program under analysis. In the body we have a switch case, which compares the system call number in register R7 with the service routine associated with it in order to call the right introspection routines which dumps it. Finally, in the epilogue we unlock the introspection code by restoring the interrupts and the preemption.

[1]Functions available in the user space library are not available in kernel but sometimes alternative kernel variant are implemented. In fact, the kernel variant of malloc are (kmalloc and vmalloc).

Listing 4: An entry of the GOT-like table.

```
sprintf:
    ldr     pc, [pc, #-4]
.word       0xc02f8ba8
```

*2) Phase 2: Process Filtering:* This phase filters out the system calls from non monitored programs by checking the PID of the caller program with the PID or COMM of the program that we are currently monitoring. The PID and COMM are stored in memory in the same structure as mentioned in Section III-A2. Note that, in External Tracing Mode we needed some optimization to minimize the overhead due to halting the CPU, introspecting the memory, and resuming. Instead, with In-kernel Tracing Mode, we can simply check the PID and COMM of every system call because the overhead is negligible (only about 10 assembly instructions).

*3) Phase 3: Memory Introspection & Argument Unmarshalling:* This phase introspects the memory in order to reconstruct the system call and its arguments, according to the value system call number present in the register R7[2]. For example, if this introspection code finds in the register R7 the system call number 4, the introspection code calls the _log_sys_write_entry routine, which reads the memory content and unmarshall according to the offset and size of the data; in this specific case, we unmarshall the content with the format string sys_write: 0x%08x %s 0x%08x though the routine sprintf (this routine is available to the introspection code because we have implemented a GOT-like table, which resolves routine symbols) to a buffer. The resulting buffer is then sent over UDP by using the kernel helper functions.

Our GOT-like table gives us the ability to use kernel helper functions. Thus, it is possible to use routines like sprintf in the introspection code without the need of relying on a compiler. For example, when sprintf is called with bl instruction, the return address is stored in the link register (lr), then the ldr instruction effectively loads the entry point of sprintf.

## IV. Implementation details

In this section we describe how we implemented OpenST by leveraging OpenOCD, a popular, open-source JTAG abstraction layer that support various JTAG adapters and target development boards. Moreover, we describe the overall workflow to process an APK using OpenST.

Our prototype comprises a host PC and a target board, connected via JTAG (through an USB adapter) and ethernet. The host PC orchestrates the analysis (e.g., launching the target application), manages the state of the target board (e.g., boot, reboot, restore), and serves as a repository of the collected system call traces.

### A. JTAG Abstraction Layer

We rely on OpenOCD to handle the communication through the JTAG interface. OpenOCD offers high-level

[2]Each system call has an identifying number [16].

primitives to low-level functionality to access the CPU state and the physical memory. In particular, for the External Tracing Mode, we modified the source code of OpenOCD to instrument existing routines. For example, we added `systrace`, a high-level function that allows the user to upload an application and trace its behavior. In implementing `systrace`, we used some of the routines already available in OpenOCD including read, write, and breakpoint-management routines (i.e., `target_read_phys_memory`, `target_write_phys_memory`, `breakpoint_*`, respectively). As mentioned before, we instrumented the callback function, which would normally just halt the target board, to automatically introspect, unmarshall, and dump the system calls, its parameters and resume the execution. The implementation of the In-kernel Dynamic Tracing Mode was simpler: We used the JTAG interface to inject (using the `write` command) the introspection code in the kernel memory.

## B. Workflow

The typical analysis workflow has three steps.

**Boot.** First, the target board boots from the SD card. More precisely, its ROM contains the various boot-loading stages. The last-stage bootloader (U-Boot) can continue the booting process "remotely." For example, it able to load the kernel binary image from a TFTP server, and mount the root file system over NFS. By leveraging this functionality, we make the target board boot an Android system that actually resides on a remote, NFS-served filesystem. This has the advantage of making the restore procedure much simpler to handle, minimizing the wearing of the NAND chips of the SD card, thus ensuring scalable deployment. Notice that, at each boot, a fresh copy of the filesystem is restored.

**Tracing.** Secondly, after the board has booted an Android system, OpenST installs the target application (as an APK file) though command line. To this end, we use the Android debug bridge (`adb`) tool, a command line utility that can be use to manage a running Android system. More precisely, OpenST uses `adb` to install APKs (`adb install file.apk`), and launch the respective application (`adb shell am start -n a.package.name a.package.name.MainActivity`). In External Tracing Mode, since the actual tracing happens on the host PC, we display the traced system calls to the standard output and/or log them into a file. In In-kernel Tracing Mode, instead, the injected instrumentation routines take care of sending the traces via UDP to the host PC.

**Restoring.** Lastly, once the target application has been analyzed, OpenST issues a reboot command through JTAG and restores the previously backed-up filesystem. This ensures that any modification made by the target (malicious) application to the filesystem do not affect the subsequent analyses.

## V. Experimental Evaluation

We have conducted experiments in order to evaluate both the resilience of OpenST to evasion techniques commonly used by malware and the overhead caused by our tracing system.

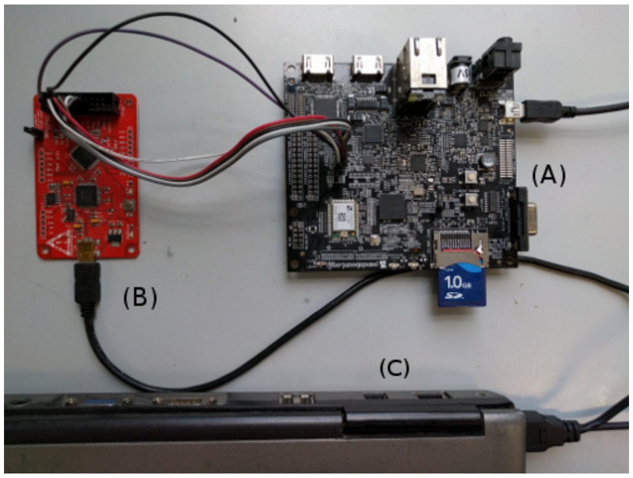


Fig. 3: The prototype of OpenST used in our experiments.

## A. Experimental Settings and Dataset

We implemented OpenST in our lab, as shown in Figure 3, using a PandaBoard ES equipped with the OMAP4460 SoC as the target board ($200), running a Linaro Android version 4.4.4 build id KTU84P, and built on Sat Oct 25 19:41:39 UTC 2014. We used the Bus Blaster v3 JTAG adapter ($50) to connect the host PC to the PandaBoard, using a standard USB cable. Moreover, the host PC is connected to the board via an ethernet cable.

We obtained access to the BareDroid dataset, which comprises nine malicious applications, including six variant of Android.HeHe [17] and one sample of Android.Pincer.A [18] that are known to use evasive techniques against emulator-based analysis environments. Note that, given the different Android versions, some of the applications were incompatible with the Linaro Android that supports the PandaBoard: We simply excluded these APK files, ending up with seven applications in total. Even though these malicious applications may rely on simple heuristics that can be made ineffective by patching the emulators, more sophisticated ones can rely on cache behavior of the emulator or sensors, which cannot be circumvent easily. However, these samples are sufficient to highlight resilience of OpenST compared to the emulator.

## B. Micro Benchmark Experiment

In this experiment we measured the execution time of the top nine most popular I/O system calls and one utility system call (i.e., read, write, open, close, getpid, mkdir, rmdir, dup, ioctl, dup2). Specifically, we measured the performance hit of OpenST compared to an unmodified environment. In order to accurately measure the time spent by the system to execute each system call, we used the CPU cycle counter, present in the ARM processors, which provide us an high precision time reference. More precisely, we read the clock counter register and then divide by the CPU frequency in order to obtain the time. We used system calls of the host PC to evaluate the time.

From the results of this experiment, summarized in Figure 4, it is clear that the External Tracing Mode is the most demanding in terms of speed overhead. However, recall that these results are *micro* benchmark. In the following section we show that the overall slowdown imposed by OpenST, albeit non negligible, is less substantial.

## C. Macro Benchmark Experiment

In this experiment we show the overall impact of OpenST on the entire execution of a traced executable file. To this end, we measured execution time from the moment the application starts (though `adb`) to the moment the application exits its main activity. Since we do not want to introduce artifacts, we did not instrument the APKs to measure time. Instead, we measured the overall time using the `time` command line utility. Table I shows the overall results.

## D. Evasion Resiliency Experiment

The main advantages of OpenST (as well as any bare-metal-based tracing approaches) is its resilience to evasive techniques used by state-of-the-art malicious applications. To asses its resilience, we conducted an experiment to monitor the behavior of a malicious application while being traced by OpenST.

We run the experiment in three different settings. In the first setting, we obtained the Android image used by the authors of BareDroid, containing the modified version of the stock Android version that they used in their paper. This Android image leverages SELinux policies to elegantly "trap" system calls, without introducing any other modifications to the system. As a result, the instrumentation mechanism itself is not detected by the malicious applications. In fact, the malware families that we used in this experiment (i.e., 6 variants of Android.HeHe [17] and 1 of Android.Pincer.A [18]) only detect the emulator, not the instrumentation technique. In other words, by using these families we guarantee that whenever we observe signs of "evasive" behavior, those must be due to the fact that the malware has detected the emulator (e.g., by inspecting the IMEI, phone number, or sensors, which have bogus/default values in the emulator). In the second and third setting, we used OpenST in both its tracing modes.

Table II compares the number of system calls observed in the first setting (emulator-based tracer) with the ones observed in the third setting (OpenST-based tracer). The table shows the total number of `read`, `write`, `open`, and `close` system call invocations traced by each approach. From the first column we notice that the malware is actively evading the emulator, because it invokes only a handful of system calls (i.e., it is hiding part of its program). With OpenST, instead, in both modes we observe a significantly higher number of system calls invocation.

TABLE I: Macro Benchmark Experiment (Section V-C). Overall time [s] spent during the macro benchmark experiment.

| Family Name | Emulator | In-kernel Tracing | External Tracing |
|---|---|---|---|
| Android.HeHe.1 | 11.5 | 10.6 | 832.7 |
| Android.HeHe.2 | 11.3 | 12.6 | 794.5 |
| Android.HeHe.3 | 10.7 | 10.7 | 902.9 |
| Android.HeHe.4 | 10.8 | 11.0 | 815.2 |
| Android.HeHe.5 | 11.8 | 10.5 | 805.4 |
| Android.HeHe.6 | 11.7 | 11.9 | 839.6 |
| Android.Pincer.A | 7.7 | 7.7 | 635.3 |
| Total | 75.5 | 75.0 | 5625.6 |

TABLE II: Number of system-call invocations (read, write, open, close) observed with an emulator-based tracer vs. an OpenST-based tracer.

| Sample | Emulator | OpenST | |
|---|---|---|---|
| | | In-kernel | External |
| Android.HeHe.1 | 3, 0, 0, 0 | 160, 187, 54, 67 | 165, 185, 54, 67 |
| Android.HeHe.2 | 7, 1, 0, 0 | 171, 202, 54, 67 | 169, 200, 54, 67 |
| Android.HeHe.3 | 11, 2, 0, 0 | 121, 143, 54, 67 | 131, 151, 54, 67 |
| Android.HeHe.4 | 2, 0, 0, 0 | 155, 190, 54, 67 | 145, 181, 54, 67 |
| Android.HeHe.5 | 3, 0, 1, 0 | 144, 166, 54, 67 | 155, 187, 54, 67 |
| Android.HeHe.6 | 3, 0, 0, 0 | 167, 215, 54, 67 | 172, 232, 54, 67 |
| Android.Pincer.A | 1, 2, 0, 0 | 67, 54, 117, 96 | 66, 55, 117, 96 |
| Overall | 30, 5, 1, 0 | 985, 1157, 441, 498 | 1003, 1191, 441, 498 |

## E. Scalability Estimation

In Figure 5 we compare the scalability of OpenST with respect to BareDroid [13] and emulator-based approach with an hypothetical budget of \$50,000, which is the same used in BareDroid [13]. To this end we use the cost model proposed in BareDroid. In particular the throughput $X$ according to the cost model used in BareDroid is:

$$X = \frac{1}{t_{exec} + t_{restore}} \cdot \frac{TotalBudget}{DeviceCost}$$

by inserting the value for OpenST we obtain the Table III.

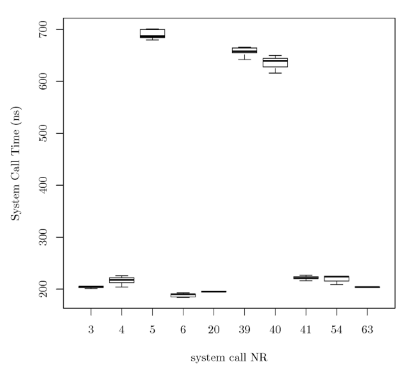
The experiment shows the resilience of OpenST in both mode. External Tracing, which does not require modification of the target OS, is 74× slower than an emulator-based approach. In-kernel Dynamic Tracing, which does require injection of introspection code into the kernel, is essentially as fast as emulator-based approach.

## VI. Discussion

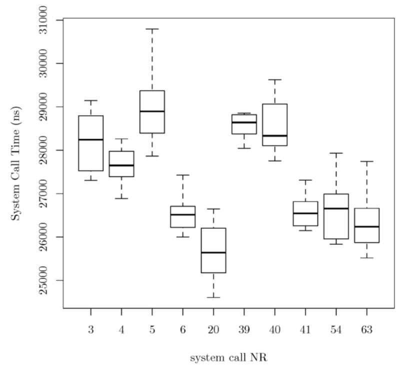
The goal of our work was to show the feasibility of implementing a low-cost, open research tool for analyzing untrusted programs at two levels of transparency (in kernel, or externally). The results of our experiments suggest that such a tool can be implemented and that, at the price of a substantial overhead, even evasive programs can be traced in a fully transparent way. Remarkably, given a budget of \$50,000, OpenST scales better than BareDroid, that is the current state of the art.

TABLE III: Value used in our cost analysis.

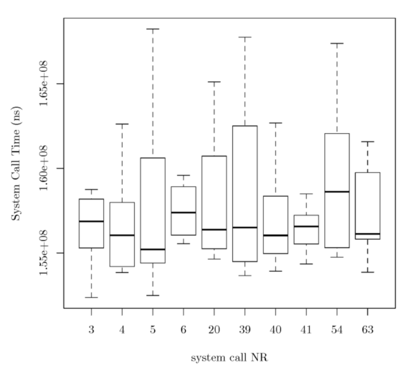
| Method | Cost per device/CPU (\$) | Restore time (s) |
|---|---|---|
| OpenST | 250 | 58 |
| Emulator | 300 | 1 |
| BareDroid | 349 | 32 |
| BareDroid (Full Restore) | 349 | 141 |



(a) Unmodified System.



(b) In-kernel Dynamic Tracing Mode.



(c) External Tracing Mode

Fig. 4: Micro Benchmark Experiment (Section V-B). These boxplot show the time (Y axis) taken by each of the 10 most popular system calls (X axis) to complete its execution.

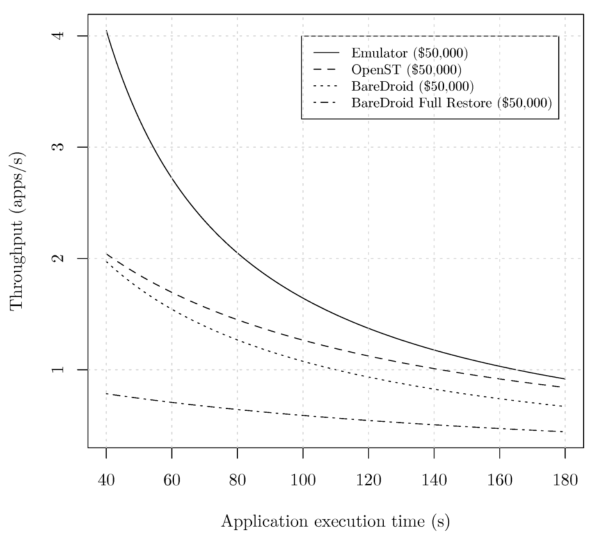


Fig. 5: Throughput of a system (Y axis) by allocating a certain amount of execution time (X axis) to an application.

**Benchmarks.** In the micro benchmark experiment we rely on CPU cycle counter in order to measure the time of system calls accurately. In fact, we calculated the time spent by a system call by dividing the cycle counted by the frequency. A drawback of this measurement techniques is that is not really portable (e.g., to an emulator). Generally, micro benchmark against emulator is not fair. In fact, emulators have overhead mostly distributed uniformly across every instruction, while in OpenST the overhead are present only during system calls.

**Debug Ports.** OpenST strongly rely on the JTAG port, which is not always accessible on all devices. A JTAG port is mostly present and well documented in develop boards. Although faster tracing ports can be used, this would impact significantly the total cost. A promising alternative would be to investigate the use of CoreSight's embedded trace router (ETR), which allows to copy the instruction trace into a memory region (properly managed by a kernel driver). However, in addition to the increased cost (ETR is only supported by a few development boards), there at least two challenges that need to be tackled. First, this technique requires a kernel driver, which obviously affects the transparency of the tracing process. Secondly, ETR performing tracing at *instruction* level, without the possibility of halting the CPU to inspect registries and memory to derive obtain the system call arguments. We believe that this approach could be combined with a JTAG-based approach, which will be in charge only of halting the CPU when needed, leveraging the fast ETR functionality for data transfer (instead of the JTAG port, which is slow).

**Automation.** OpenST's In-Kernel Tracing Mode requires that the instrumentation code is written (either in assembly or in high-level language, then compiled in assembly). This process can be automated by parsing the DWARF information as we did in the External Tracing Mode. However, it required us time to manually analyze each system call implementation, in order to understand how the arguments are received by the called function, and how to properly handle their de-serialization.

## VII. Related Works

OpenST is related to the recent approaches that proposed emulator or bare metal-based introspection mechanisms.

CopperDroid [5] is an emulator-based tracer built on top of QEMU. The framework is capable of tracing both Android and OS-level calls. It utilizes VMI techniques to inspect the internal structure of the emulator. The major difference between OpenST and CopperDroid is the underlying system. CopperDroid is based on an emulator, OpenST is based on bare metal.

BareBox [10] is an VM-aware platform that implement fast restore system. It has a Meta-OS running alongside with the target OS to manage snapshotting and restoring of both memory and disk. In addition it offers system call tracing by hooking the System Service Descriptor Table (SSDT). The major difference between OpenST and BareBox is where introspection is conducted. BareBox does the introspection inside the target system (similary to our In-kernel Dynamic Tracing Mode). However, OpenST's External Tracing Mode is profoundly different from (and more transparent than) BareBox.

BareDroid [13] is a device-based platform for analyzing Android malware. The system is designed to be efficient, in fact only the first partition (boot partition) is rewritten at each analysis, while the rest of the partition is written if the boot integrity chain fails. The major difference between OpenST and BareDroid is the trusted code base. In BareDroid, the kernel is assumed to be not compromised. Instead, with in-kernel mode we do not require any modification to the kernel, because we patch it dynamically when neede by the analysis.

In [19] the author implemented a Windows kernel driver able to detect sign of exploitation. In particular, the kernel driver is able to read the Hardware Performance Counters (HPCs) of modern processors. This allows to sample the performance of application. In their experiment, they show accuracy of detecting 99.5% shellcode exploits. In OpenST, we use hardware-level CPU features, uet with a different goal (i.e., tracing) than in [19]. In fact, we believe that [19] is complementatry to OpenST.

LO-PHI [20] contributed in the direction of hardware-based taint analysis for increasing the transparency, making it possible to analyze applications that use evasive techniques. In particular, LO-PHI is capable of passively sniffing the disk activity at SATA level, by bridging it though a double SATA port of an FPGA based board. Additionally, LO-PHI is capable of dumping the memory content by constantly pooling the physical memory at high rate though the PCIe interface.

## VIII. Conclusions

With the increasing number of applications developed for Android, it becomes necessary to develop automated tools for analyzing them in order to identify the possibly malicious ones. These analysis tools typically perform dynamic analysis of the applications by using emulator-based sandboxes. It is well known that emulators do not faithfully reproduce every aspect of the hardware, and that they leave artifacts exploitable by malicious applications that can evade the analysis by concealing their malicious behavior, once they realize that they are executed in an emulated environment.

In this paper we addressed this problem by proposing OpenST, an hardware based tracer. We have described in detail its design and implementation. We have validated OpenST by successfully tracing malware that are known to employ evasion techniques. We have carefully evaluated the tracing overheads, showing a trade-off between a completely transparent, external tracing approach (although with severe performance impacts), and a slightly less transparent in-kernel approach with an overhead which is comparable with other state-of-the-art techniques, while still improving upon their stealthiness.

We believe we demonstrated that OpenST can provide automatic support for the dynamic analysis of malware that employs evasion techniques, and that otherwise would need to be manually inspected.

## References

[1] IDC. Smartphone os market share. [Online]. Available: http://www.idc.com

[2] T. Micro. Security predictions: The fine line. [Online]. Available: http://www.trendmicro.com

[3] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket." in *NDSS*, 2014.

[4] V. V. der Veen and C. Rossow. (2015) Tracedroid.

[5] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors." in *NDSS*, 2015.

[6] L. K. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 569–584.

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

[8] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 447–458.

[9] Y. Jing, Z. Zhao, G.-J. Ahn, and H. Hu, "Morpheus: automatically generating heuristics to detect android emulators," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 216–225.

[10] D. Kirat, G. Vigna, and C. Kruegel, "Barebox: efficient malware analysis on bare-metal," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 403–412.

[11] ——, "Barecloud: bare-metal analysis-based evasive malware detection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 287–301.

[12] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 55–69.

[13] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna, "Baredroid: Large-scale analysis of android apps on real devices," in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 71–80.

[14] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, vol. 41, 2009, p. 86.

[15] A. I. Centre. Cortex-a9 technical reference manual. [Online]. Available: http://infocenter.arm.com

[16] F. Electrons. Linux cross reference. [Online]. Available: http://lxr.free-electrons.com/source/arch/arm/kernel/calls.S

[17] F. Blogs. Android.hehe: Malware now disconnects phone calls. [Online]. Available: https://www.fireeye.com

[18] F-Secure. Android.pincer.a. [Online]. Available: https://www.f-secure.com

[19] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Research in Attacks, Intrusions and Defenses*. Springer, 2014, pp. 109–129.

[20] C. Spensky, H. Hu, and K. Leach, "Lo-phi: Low-observable physical host instrumentation for malware analysis," 2016.