Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging

Marcus Botacin, University of Campinas
Paulo Lício de Geus, University of Campinas
André Grégio, Federal University of Paraná

Malware and code-reuse attacks are the most significant threats to current systems operation. Solutions developed to countermeasure them have their weaknesses exploited by attackers through sandbox evasion and anti-debug crafting. To address such weaknesses, we propose a framework that relies on modern processors' branch monitor feature to allow us to analyze malware while reducing evasion effects. The use of hardware-assistance aids in increasing stealthiness, a key feature for debuggers, since modern software (malicious or benign) may be anti-analysis armored. We achieve stealthier code execution control by using the branch monitor hardware's inherent interrupt capabilities, keeping the code under execution intact. Previous work on branch monitoring have already addressed the ROP attack problem, but require code injection and/or are limited in their capture window size. Therefore, we also propose a ROP detector without these limitations.

CCS Concepts:• **Security and privacy** → **Malware and its mitigation; Software reverse engineering;** *Information flow control;*

General Terms: Malware, ROP, Debug, Branch Monitor

Additional Key Words and Phrases: Malware, ROP, Debug, Branch Monitor

ACM Reference Format:
Marcus Botacin, Paulo Lício de Geus, André Grégio, 2017. Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging. *ACM Trans. Priv. Secur.* 21, 1, Article 4 (January 2018), 36 pages.
DOI: 10.1145/3152162

1. INTRODUCTION

Malware authors continuously improve their code to thwart detection by evading analysis environments, such as sandboxes and debuggers. Both legitimate software (for intellectual property protection) and malware (for detection avoidance) may be equipped with anti-analysis and/or anti-debugging techniques, causing the need for increased stealthiness to overcome these techniques and so perform more dependable malware analysis. Along with malware infection, code injection used to be one of the main attack vectors to subvert systems functioning. The adoption of non-executable pages supported by hardware (No eXecute - NX/eXecute Disable - XD) and data execution prevention (DEP) eliminated this problem in practice. However, attackers found another way of leveraging control flow deviation by chaining blocks of code (gadgets) through `RET` instructions. This is known as Return-Oriented Programming (ROP) and is cur-

This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, Universal 14/2014, process 444487/2014-0) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).
Author's addresses: Marcus Botacin and Paulo de Geus, Institute of Computing (IC); André Grégio, Department of Informatics (DInf)

rently the main vector for injection attacks. Recently, techniques based on Control Flow Integrity (CFI) and code length arose to counter such attacks with reasonable effectiveness.

Based on the aforementioned issues, we introduce a hardware-assisted solution to address sandbox evasion and anti-debugging equipped malware in a stealthier way, and to detect ROP attacks in real time while overcoming limitations of existing state-of-the-art, hardware-assisted approaches. In summary, we make the following contributions:

(1) **Current threats and solutions scenario review**: we present a review on the threat landscape scenario and current countermeasure and analysis tools, discussing their weaknesses. Specifically, we review transparent analysis solutions as well as branch-based ones.
(2) **Branch monitoring framework**: we propose a complete, modular framework based on hardware monitoring features, allowing for further applications that overcome current and future state-of-the-art limitations and weak points. The framework solution is open source, being available on Github[1]. Media is also available, on Youtube[2].
(3) **Low-overhead malware analysis**: we leverage our framework to build a malware analysis tool with lower development efforts than the current state-of-the-art ones. As far as we know, no other malware tracer is based on such kind of monitoring.
(4) **Stealthier, granular debugger**: we demonstrate how to implement granular debugging based on our framework without using single-step flags, increasing the stealthiness against evasive malware. Again, we have no knowledge of other debugging solutions based on branch monitors.
(5) **ROP attack detector**: we present an improved implementation of current ROP detection heuristics, based on our framework, which does not require code injection, a limitation on other approaches.
(6) **Hardware Improvements**: We suggest possible hardware enhancements for branch monitors based on the challenges we faced when developing our solution.

The remainder of this paper is organized as follows: in Section 2, we define the threats to be addressed, review the current threat and countermeasure scenario, and introduce the hardware feature used in our solution; in Section 3, we review the state-of-the-art solutions and pinpoint their weaknesses; in Section 4, we state the basis for our framework; in Section 5, we present solutions developed upon our framework; in Section 6, we discuss our contributions, proposal limitations and future developments; finally, in Section 7, we present our conclusions.

2. BACKGROUND AND THREAT MODEL

2.1. Malware analysis and evasion

Techniques for malware analysis are usually classified as static or dynamic [Sikorski and Honig 2012], each one with its own limitations. Static analysis may be susceptible to both theoretical (e.g., opaque constants [Moser et al. 2007]) and practical (e.g., packing, encryption, and obfuscation [Gao et al. 2014]) limitations. Dynamic analysis is often employed as an additional analysis layer in order to overcome such limitations [Egele et al. 2008], relying on the sample's execution in a controlled environment (sandbox). The execution usually happens on an emulator, due to instrumentation is-

[1]https://github.com/marcusbotacin/BranchMonitoringProject
[2]https://www.youtube.com/watch?v=BguVzqMt_j0&list=PLVYZ2jULLUDvqFVpU3pCZGlY9gCzYoyXP

sues, or in a virtualized environment, due to scalability issues. Code execution in a non-native environment may be used by malware to detect an analysis environment and thus to hide its malicious intent. Environment detection is performed mainly through fingerprinting or side-channel effects on instruction execution [Marpaung et al. 2012], since emulator implementations do not exactly look like physical processors. Currently, there are tools to automatically detect such side effects [Shi et al. 2014].

Many authors tried to address evasive malware issues, either by detecting their split personalities under an emulator [Balzarotti et al. 2010] and counter-measuring its side effects [Vasudevan and Yerraballi 2006b], or by running the sample on a bare metal environment [Kirat et al. 2014]. Trying to mask execution side effects is an ineffective solution since it can only assure the execution of samples which employ known detection tricks. Regarding this scenario, a non-evadable malware analysis tool should be able to run code in a native processor, which is called transparent execution.

Bare metal systems often present another issue related to detection: their intrusiveness over the traced sample. Systems which rely on techniques such as DLL injection can be detected by hashing sample's own memory. For this reason, higher-privileged monitoring tools are required [Rossow et al. 2012]. However, as the operating system (OS) evolve, high-privileged instrumentation becomes harder due to new security mechanism – Windows Kernel Patch Protection (KPP)[3], for example, denies kernel hooking, a frequent approach for API interception. This way, a non-intrusive instrumentation is a requirement for bare metal malware analysis on modern OS.

2.2. Debuggers: requirements and implementations

Debuggers can be used to assist malware analysis and reverse engineering, allowing the investigation of several execution paths. In general, a debugger should provide:

— **Small Step Execution**: A debugger should allow for region of interest inspection in a granular way—from single step to function call trace.
— **Breakpoint Information**: A debugger should assure that the breakpoint region is known. In other words, it should provide predictability to the execution. The combination of the two aforementioned requirements matches the context requirement [Rosenberg 1996]. Due to the latter, probing approaches are not suitable for debuggers.
— **Context Inspection**: Given a breakpoint, the debugger should be able to retrieve information about the current execution context, such as memory and register values and/or function called.

Current debuggers are built on top of distinct techniques—OS support, emulation or injection, and hardware support. Most OSs provide interfaces that allow process control. Some Unix-like systems, for instance, provide the *ptrace* API, which is the base for tools like the `strace` tracer and the `GDB` debugger. This solution is not transparent since the presence of the tool can be discovered with the tool itself(`if (ptrace(PTRACE_TRACEME, 0, NULL, 0) == -1)`, then it is detected). Windows also provides its own debug facilities[4]. However, they are also not transparent, being detected by the `IsDebuggerPresent` API[5]. Other debug solutions rely on hardware features, such as the `step-by-step` execution defined by setting a `trap flag` in a `debugctl` MSR register, which can be detected by samples through reading that register.

[3]http://technet.microsoft.com/pt-br/library/cc759759(v=ws.10).aspx
[4]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679303(v=vs.85).aspx
[5]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680345(v=vs.85).aspx

2.3. ROP attacks

Code injection used to be one of the main attack vectors to subvert systems functioning. The adoption of non-executable pages supported by hardware (NX/XD) and data execution prevention (DEP) eliminated this problem in practice. However, attackers found another way of accomplishing control flow deviation by chaining blocks of code (gadgets) through `RET` instructions. This is known as Return-Oriented Programming (ROP) and is currently the main injection vector. This kind of attack is not prevented by existing mechanisms since the code chains are not composed of any externally-injected material but perfectly legitimate code in system memory. In a general way, the chains are composed of a few instructions, which when combined and properly chosen may satisfy the attacker's desire to execute specific, arbitrary computation [Göktaş et al. 2014].

Recently, techniques based on Control Flow Integrity (CFI) and code length arose to counter such attacks with reasonable effectiveness. However, many issues are still posed by them, such as recompilation or code-injection.

2.4. Performance monitoring

Modern processors have many sensors on their platforms which allow for monitoring performance event indicators. These sensors may also be used for other purposes, such as security ones, as proposed in this work. Each vendor presents their own set of monitors—Intel [intel 2015], AMD [AMD 2012], and ARM [ARM 2011]. Due to availability issues, this work is based on the Intel platform. It is composed of two sub systems: PEBS (Precise Event Based Sampling) and LBR/BTS (Last Branch Record/Branch Trace Store). The first is responsible for collecting information about general system events, such as instructions retired, cache hit/misses, branches predicted and so on. The second is responsible for monitoring control flow deviations. It can store both source and target addresses of deviation instructions. Despite being called branch monitors, they can monitor any control flow deviation instruction, including `CALL`, `RET` and exceptions, beside ordinary branch ones (`JMP,JNE`). For more details, see Appendix A.

Both schemes have two storage options: register and memory-based. The first option allows the system to store data in a limited number of specific purpose MSR registers, whereas the second one provides an unlimited storage in OS memory pages. The branch monitoring mechanism is named LBR when operating in the first mode and BTS in the second. Collecting data in MSR registers requires polling, which may cause data loss in the LBR case. The memory-based approach for PEBS and BTS, in turn, have the advantage of having the ability to generate an interrupt when a given threshold is reached; this way no data is lost by the capturing mechanism. These schemes are activated by setting special flags in MSR controls and impose theoretically zero overhead, since they are processor features. For MSR access, a kernel driver is required. In addition, as a processor feature, a physical machine is required, since virtual machines do not emulate such special MSRs. The mechanisms operate in a system-wide manner. Therefore, no process information is available for filtering. The LBR mechanism, however, is able to filter deviation types (branch, call or ret). Both LBR and BTS can filter data capture level—kernel or userland.

2.5. Threat models

The assumptions presented here will guide the solution's development and evaluation. We have the following threat model for malware analysis and debug scenario.

- **Evasive malware**: we target samples with virtual machine detection as the anti-analysis mechanism.

— **Userland threats**: we assume that samples will execute in userland and no kernel activity is performed. This assures the integrity of our kernel driver, as stated by Rossow et al. Monitoring userland threats is a frequent assumption on malware sandboxes. It can be seen in solutions like Cuckoo Sandbox [Guarnieri 2013] and CWSandbox [Willems et al. 2007].
— **Stealthier analysis**: we assume that our solution will be running on a physical machine with performance monitoring support.
— **System API usage**: we assume that sample-OS interactions are performed through default system APIs. This allows us to introspect system addresses in order to reconstruct sample flows.
— **Modern OS**: we assume that the execution environment is a modern OS, where kernel instrumentation is denied by modern protection mechanisms.

Below, we show the threat model for the ROP scenario.

— **Return-based**: we assume that attacks are based on return gadgets, ignoring other forms of code reuse attacks, such as Jump- or Loop-oriented ones.
— **Unobtrusive monitoring**: we aim to monitor software without any code injection or emulation.
— **No prior information**: our solution is aimed to monitor any binary in the system without additional information about it.

3. RELATED WORK

Transparent malware analysis. Currently, two main kinds of techniques are employed in transparent malware analysis: HVM (Hardware Virtual Machines) and SMM (System Management Mode) instrumentation. HVM are systems which rely on virtualization instructions available in modern processors—Intel VT-x and AMD SVM—to build a transparent environment, since these technologies allow running code on the physical processor. In addition, they offer instrumentation facilities, such as double page translation mechanisms. The transitions from root to non-root mode also provide a way to monitor system events. Systems like the malware tracers Ether [Dinaburg et al. 2008] and MAVMM [Nguyen et al. 2009] make use of this technique to build their transparent systems. SMM mode, in turn, is a specific processor mode to manage the system at a very low level. It consists of an executable portion of code resident in the system BIOS, triggered by special interrupts called System Management Interrupt (SMI). This mode is well isolated from other execution modes by address redirection. It allows transparent execution since it is bare metal based. Systems like MALT [Zhang et al. 2015] and SPECTRE [Zhang et al. 2013] make use of SMM instrumentation to transparently monitor systems. One main disadvantage of HVM and SMM approaches is their development complexity: HVM requires writing an instrumented hypervisor. In some cases, such as in MAVMM, a hypervisor has to be built from scratch, since a minimal Trusted Code Base (TCB) is required. SMM, in turn, requires rewriting BIOS code—a task allowed only on unlocked systems. In addition, such systems cannot rely on any library, given their low-level placement. Another issue is the overhead imposed by the instrumentation routines. HVM exits may impose an overhead of the same magnitude as the system execution's, such as on Ether's case [Dinaburg et al. 2008], therefore being impractical for some uses. Our solution intends to be a lightweight version in the same line as these approaches, allowing native code execution and low level inspection but with a significant reduction in overhead and development efforts.

Debugger. Most of the current debugger developments are focused on complex software architectures, such as high level constructions [Chiş et al. 2015], distributed systems [Mäkelä et al. 2013; Schulz and Mueller 2000; Ho et al. 2004], and GPU programming [Sharif and Lee 2008]. However, few efforts were made toward a debugger more resistant to evasive malware. The closest attempts to build such debugger as proposed in our work resulted in MALT [Zhang et al. 2015] and HyperDBG [Fattori et al. 2010], which employ SMM introspection and HVM, respectively. Our work was intended to be a lightweight version in the same line as these approaches, requiring smaller costs of development and performance, although it implies on some restrictions, such as analysis inside the kernel and therefore requiring protection to avoid kernel subversion.

ROP attacks. ROP detection approaches can be classified in compilation-time, instrumentation, binary-rewriting and run-time. Approaches based on compilation time such as control flow locking [Bletsch et al. 2011a; Onarlioglu et al. 2010] exploitability aim to avoid vulnerable constructions generation, thus reducing ROP exploitability. The main disadvantage of this approach is that it cannot be applied to existing binaries. Instrumentation approaches [Davi et al. 2009; Chen et al. 2009; Graziano et al. 2016] are applicable to existing binaries without recompilation. These solutions aim to detect exploitation in real-time. However, they suffer from limitations of the instrumenting tools they are built on. Binary rewriting solutions [Hiser et al. 2012; Pappas et al. 2012; Wartell et al. 2012] can also be applied to existing binaries and do not require instrumentation. They rewrite the binary on first execution, hardening it against exploitation. The main disadvantage of this approach is its limitation to handle dynamically generated code. A broader approach is to leverage hardware monitors in order to inspect application flows. This approach can be applied to existing binaries and do not rely on emulated environments. Hypercrop [Jiang et al. 2011], for instance, leverages HVM to identify ROP attacks. However, its overhead is prohibitive. A lightweight approach to hardware assisted monitoring is to leverage performance counters. ROPecker [Cheng et al. 2014] and KBouncer [Pappas et al. 2013] make use of the LBR mechanism to identify and counter ROP attacks. ROPecker and KBouncer are the closest related to ours in the scope of ROP detection. However, they present some limitations, such as using the limited LBR instead of the BTS storage and requiring DLL injection. Our work is intended to overcome such limitations.

Branch Monitoring. Distinct solutions have been deployed using performance monitors in a general way. Beside the ones on ROP detection, they were applied in other attacks evaluation. [Yuan et al. 2011] relies on performance data provided by Linux Perf[6] in order to identify attacks. [Kompalli 2014] works in a similar way but its underlying tool is Intel Vtune[7]. Both solutions, however, are intended to detect system misbehavior in a general way, whereas our proposal is to trace specific process activity. In this sense, the work closest to ours is [Willems et al. 2012], which is able to rebuild some traces from a program crash. However, this solution is not aimed at Control Flow Graph (CFG) reconstruction or debugging. It is also limited to using the few LBR registers. Our work solves it and provides a broader solution to malware tracing.

The usage of LBR and BTS. Most solutions based on branch monitoring make use of the LBR mechanism instead of the BTS one. This way, many comparisons made in this work consider only LBR, such as on ROPecker's and KBouncer's study cases. CFIMon [Xia et al. 2012], for instance, makes use of BTS to enforce a CFI policy. Unlike this work, it uses a 2-phase heuristics, which requires binary prior-analysis. This way, it is not directly comparable to this work, which implements a 1-phase policy, such as in KBouncer. [Aktas and Ghose 2013] is the only one we are aware which addresses the specific usage of BTS for security purposes. In such work, BTS is used to validate

[6]https://perf.wiki.kernel.org
[7]https://software.intel.com/en-us/intel-vtune-amplifier-xe

control flow path transitions. The work hereby presented, however, differs from it in many ways, since our goal is to implement security tools, such as malware tracers and debugging facilities, whereas the cited work is concerned only with implementing runtime policy enforcement. Its syscall trace system is more focused on abnormal behavior detection than on tracing a given binary itself, which is proposed by us. Given these differences, we are able to provide more flexible solutions, which allows us, for instance, to reduce overhead. Moreover, our solution presents the same advantages of the aforementioned approach, such as not requiring binary modifications. [Soffa et al. 2011] makes use of both LBR and BTS in order to discuss the application of hardware monitors in the context of software engineering. Although this work had already suggested branch monitor use in order to track executed paths, it can be considered as a first step, since many aspects needed to be further discussed, such as: i) external library inspection; ii) branch-capture granularity; iii) process isolation; iv) implementation aspects. In this scope, the hereby presented work can be considered as a second-step, discussing these missing points and presenting real-world sample evaluation.

4. PROPOSED FRAMEWORK

The framework is general in nature and can be applied for collecting and evaluating control flow deviation data, in particular by the applications developed as part of this work, which implements extensions to the framework in order to apply security policies. We are mainly concerned with tracing program paths, so we adopted the branch monitor subsystem of the performance monitor hardware as our base mechanism. Given that it is able to collect the address of executed instructions, we can reconstruct the whole scenario of code execution over binaries, libraries and function calls through introspection.

To avoid losing instruction data, we opted for the BTS mechanism instead of the LBR one. The advantage relies on the ability to make use of interrupts on our system, which assures its state is coherent at inspection time. We defined a 1-instruction threshold; this implies the system will be interrupted at each control flow deviation instruction, therefore warranting the precise identification of which process is executing such instruction. This way we can easily and stealthly filter process actions, even though the BTS system itself is unable to provide such information. Conversely, solutions using LBR require intrusive hooks to provide similar capabilities, such as in [Akao and Yamauchi 2015].

Our system architecture is designed as a client-server one, in which the kernel driver is responsible for managing BTS data (server) and the user-land application for applying policies on the collected data (client). Data collection may be synchronous or asynchronous, in order to best fit a given policy. Data is collected in a system-wide fashion as it is provided by the BTS mechanism and filtered in the user-land client, so that distinct policy implementations are allowed, as shown on Figure 1.

The proposed architecture does not require any injection or interaction with the analyzed process, as data is collected by the processor and processed by distinct, independent pieces of software. It also runs in a more hidden way from evasive malware, since instructions are executed on the real processor. This framework is also easier to implement than literature-based “transparent” approaches: it requires only a kernel driver and additional user-land software, without the need of writing a hypervisor or rewriting the BIOS.

In the next subsections, we cover details about the framework implementation and the characteristics that make it flexible. It was implemented on 32- and 64-bit versions of Windows 7 and 8.

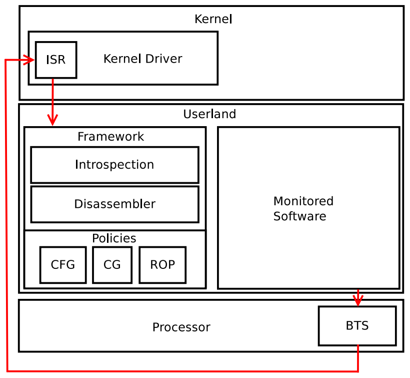


Fig. 1: Proposed Architecture. The processor fetches branch instructions from the monitored code, which triggers the BTS threshold. The raised interrupt is handled by an ISR at a kernel driver. The captured data is sent to the userland framework where introspection and disassembling are performed and policies are applied.

4.1. Driver: all about the basis

We need to access the `debugctl` MSR register, thus requiring the deployment of a driver. This driver is also required for allocating and supplying OS memory pages to the BTS mechanism, as well as providing the Interrupt Service Routine (ISR) to handle BTS interrupts and the I/O routines which send data from kernel to user-space. Each of these features are detailed next.

4.1.1. Handling Interrupts. Interrupt handling is the most important step of data collection, because this is where data preservation is effectively performed. As BTS data is stored in memory pages, we can collect it by simply using a pointer. Installing the ISR, however, is the hard part: the BTS interrupt mechanism looks into the Local Vector Table (LVT) to find out how to deliver the interrupt. The LVT defines if it is delivered through an SMI, NMI or fixed mode (the default option). In the latter case, an entry into the Interrupt Description Table (IDT) is performed. The ISR address is placed on the corresponding position of the IDT. On Windows systems, the defined IDT entry may already be allocated to another portion of the system. Changing the LVT vector offset could be an option but, in our tests, no IDT entry was available. Hooking IDT is not an option anymore on newer Windows versions since the Patch Guard mechanism prevents it.

Hooking the specific performance handler could be an option—it could be done by calling the `_HalpSetSystemInformation()` from `HalDispatchTable` to change the `HalpPerfInterruptHandler`—but this may present side effects. A non-hooking solution is to change the delivery mode on LVT to Non-Maskable Interrupt (NMI), a high priority interrupt originally aimed at exception checking. As an NMI interruption is immediately handled, it is a good choice for our monitoring goal. The NMI ISR is registered using the `KeRegisterNmiCallback`[8] function. When an NMI happens, process execution is suspended so that we can correctly retrieve its PID. It is performed by using the `PsGetCurrentProcessId`[9] function. This information, along BTS branch data, are stored in a queue, detailed below, to be collected by an I/O call. Finally, the ISR is also responsible for re-enabling the BTS interrupt mechanism, since it is disabled as soon as an interrupt happens.

[8]https://msdn.microsoft.com/en-us/library/windows/hardware/ff553116(v=vs.85).aspx
[9]https://msdn.microsoft.com/en-us/library/windows/hardware/ff559935(v=vs.85).aspx

4.1.2. Handling Data. When an interrupt occurs, the BTS buffer is full, so we have to copy its data to some other place in order to free the buffer and re-enable the monitor. In our solution, BTS buffer entries are copied to a Windows kernel list, pushed in a FIFO basis, in order to keep proper order. If the event buffer were not saved and the monitor just re-enabled, entries would be overwritten, thus falling back to the same effect of using the LBR monitor.

4.1.3. Performing I/O. The client receives data from kernel through I/O routines, with each application presenting distinct requirements for data collection. Applications that are intended to provide real time monitoring may require an asynchronous I/O in order to get results as soon as possible, whereas tracing tools may get delayed data through synchronous I/O. Each mode is detailed below:

Synchronous I/O. In this mode, ISR collected data is enqueued on a circular kernel list[10] in a FIFO way. Data is transferred to user-mode through IRPs[11] generated from a ReadFile[12] call, since the driver handle is opened as a file. The client periodically asks for more data through polling the ReadFile call. As the queue follows the FIFO rule, the data corresponds to ordered events. In this mode, BTS data will generate data at a rate higher than the consumption by the polling client. However, no data is lost since it is moved from the BTS entry to the kernel list. The driver client, however, should define a compatible polling frequency in order to not exhaust kernel memory. In our tests, a second-long interval was enough.

Asynchronous I/O. In this mode, the collected data is not stored in a queue, but on a single structure, since it is expected to be consumed as soon as it is retrieved. Once an interrupt occurs, the driver fires a previously cached I/O request in order to alert user-mode code that the requested data is available. The client is then responsible for releasing the I/O routine and collecting the stored data. The client should first release the I/O since an interrupt is intended to be fast, being protected from locking by a timer watchdog. This collection mode is named Inverted Call, since it is fired from the kernel. Notice that the client must have distinct threads for immediately handling the kernel call and processing the data independently, thus not blocking the ISR and preventing data loss that would otherwise occur due to cumulative BTS data production.

4.1.4. What happens after an interrupt. It is natural to think that interrupts will keep being raised during the ISR, which would overload the system. However, the BTS mechanism has some filtering features which help us deal with this. The main one is the execution level filter, which allows us to disable interrupts generated by the kernel, thus no branch generated by the ISR is captured.

4.1.5. Handling monitor branch data. As the BTS captures branches in a system-wide way, the client-generated branches could also be captured by the client itself. A direct way of preventing such behavior is to run the client on a core distinct from the one the monitor is running on. An alternative approach would be to perform PID tracking in the kernel.

4.2. Clients: where the magic happens

The user-land clients are the active analyzers of our system. They are responsible for retrieving the data collected from the driver and applying their policies. They can be built with complete independence from the drivers. To exemplify this claim, we have

[10]https://msdn.microsoft.com/en-us/library/windows/hardware/ff563802(v=vs.85).aspx
[11]https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets
[12]https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467(v=vs.85).aspx

implemented clients both in C and Python. The clients are responsible for keeping system information in memory and to use them for the analysis process. The basic information processing they perform are introspection and context retrieval mechanisms, which aim to enrich the raw data collected. We detail both below.

4.2.1. Introspection. The information provided by the BTS mechanism (addresses) have very little meaning in the context of a program execution. These addresses must be translated into high level constructs so that analysts could gain more knowledge about the system state. As instruction addresses point to the main memory, we can identify which loaded modules are resident in each memory region. The loaded modules may be the main binary or dynamic library code images; in the latter case, we can dig into their structure to identify known function addresses/offsets, giving information about which functions were called and/or what is being executed. The same could be done for binary images if we had debug symbols, which is not usually the case.

Code images in memory can be enumerated by using the `EnumProcessModules`[13] function. Each of their base locations can be retrieved with the `GetModuleHandle`[14] function. However, code images change their placement at every system startup due to the Address Space Layout Randomization (ASLR) mechanism (for more details, see Appendix B). Therefore, as we run on a bare metal system[15], which requires rebooting for state restoring, this code image address enumeration procedure should be repeated at every boot, thus being intrinsically ASLR-aware. If the intended usage scenario is not a sandbox, which requires rebooting, one may just repeat this procedure before every process invocation in order to get per-process, ASLR-aware data.

Given a BTS-provided address, we can identify the corresponding library it refers to by looking to the closest base address previously retrieved from module enumeration. By looking to the difference between the base address of a given library and the address pointed to by the BTS, we can compute an offset, which is mapped to a library internal function, thus leading to a higher level semantic construct. Function offsets can be obtained by inspecting host libraries through the use of tools like `DLL Export Viewer`[16] (Appendix C). The whole introspection process is illustrated in Figure 2. It is important to notice that such offset extraction occurs automatically before analysis begins by considering a list of module names supplied by the analyst. Once the extraction is performed, an offset database is created. We are able to reuse such data since, unlike module addresses, function offsets do not change due to ASLR.

4.2.2. Looking into memory. In addition to looking into what an address represents, sometimes it is useful to look to the address content—it can be executed instructions, as directly pointed to by the BTS mechanism, or function arguments, given by an introspection process. Given a memory address, the contents can be retrieved by using the `ReadProcessMemory` API[17]. It is worth to note we are allowed to perform such memory read since our framework execute with administrative privileges. It is also important to notice that memory reads, unlike writes, allow us to decrease the chance that malware discovers the framework. The ability to read memory allows us to read instruction bytes, which can be used to enrich the software analysis. However, the bytes need to be translated into a higher-level construct in order to be understood, i.e., instruction opcodes. Given an instruction address, we can easily get the opcode from

[13]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms682631(v=vs.85).aspx
[14]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683199(v=vs.85).aspx
[15]By bare metal we mean a physical machine running an actual processor, with no emulation or virtualization.
[16]http://www.nirsoft.net/utils/dll_export_viewer.html
[17]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553(v=vs.85).aspx

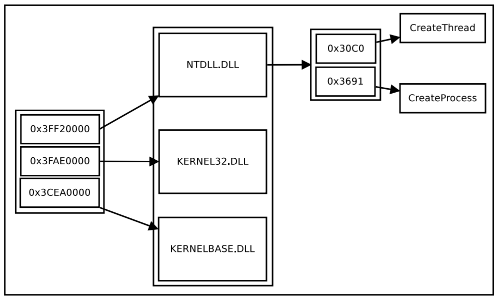


Fig. 2: Instrospection mechanism: from raw addresses to functions.

the instruction represented by the first byte of the memory dump by using a simple table. For additional details regarding the opcodes, see Appendix D. Notice that the same opcode may have slightly different meanings according to the following bytes (addressing modes). However, our solution does not need to look to these immediate values to calculate addresses, since the branch monitor provides the already calculated target addresses. This way, we can only look to the first byte and thus speed up some kind of instruction interpretation, as is required for the ROP detector through the CALL-RET CFI policy.

Despite disassembling only one byte, we may also face the scenario in which a block of code is provided. As x86 instructions are not fixed-size, we need to find out if the following bytes are immediate arguments or a new, following instruction. The disassembly of multiple instructions in our system is performed by two libraries: Pybfd [Groundworkstech 2016], a Python interface for `libopcodes` on Linux, is used for offline processing whereas Capstone [Capstone 2016], a Windows disassembler, is used for real-time processing.

Besides knowing how to interpret instructions from a memory dump, we need to know how to retrieve addresses from branch information. The straightforward dump of the first byte indicated by some branch data is not able to identify all instructions executed. That would require additional data, which can be obtained by looking to two consecutive branches. The destination address of the first branch indicates a place where the execution will start; the source address of a consecutive branch indicates that the code execution left the block at that point. As no other branch may have occurred, all intermediate instructions were effectively executed. Therefore, reading memory starting on the first address up to the second leads to all executed instructions. Figure 3 illustrates it with data from Listing 1. The execution flow enters a block of code at the first branch target address (`0x48ff5ab8`) and leaves it on the source address of the next taken branch (`0x48ff5ac0`). As no other deviation occurred, all instructions stored in that range were effectively executed. The opcodes of such instructions are identified through the disassembly of 8-bytes (the exit address minus the entry address) starting from the entry address.

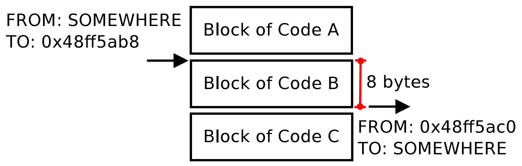


Fig. 3: 8-byte-block identification from two consecutive branches.

Listing 1: Block identification—from `0x48ff5ab8` to `0x48ff5ac0`

```
PID: 4876 FROM: 0x48ff5ab0 TO: 0x48ff5ab8
PID: 4876 FROM: 0x48ff5ac0 TO: 0x48ff5ad0
```

4.3. Validation

To validate the correct functioning of our framework, we have implemented the same solution under Intel's PIN and compared both results (Appendix E).

5. APPLICATIONS

In this section, we present security applications built upon the presented framework. Each of them makes use of a distinct feature from it, exemplifying distinct ways branch monitors can be applied for security purposes. For the sake of readability, since capturing data at branch level produces huge amounts of data, we present CG and CFG reconstruction based on minimal examples. However, the evaluation tests presented in further sections, such as 5.1.4 and 5.2, are based on real samples. The complete logs for such samples can be found on the project page [18].

5.1. Malware Tracer

Traces can provide information about malware behavior and its interaction with the system, which can be used to group similar samples, develop anti-virus vaccines, patch vulnerabilities, and so on. Our malware tracer follows directly from the data obtained from the BTS mechanism, as instructions are supplied. We have implemented two analysis features in our prototype: a call-graph viewer and a Control Flow Graph (CFG) rebuilder. The first allows for identifying a sample's behavior whereas the second can provide granular information about executed instructions, which allows heuristics development like one based on tainting.

5.1.1. Call Graph. The CG represents function calls and their relationships. To exemplify the CG visualization application, we will rely on a simple code whose host process was named `NewToy.exe`: `scanf("%s",val); printf("%s\n",val);`. `CALL` and `RET` instructions are directly captured by the BTS mechanism and function identification is performed by the previously mentioned introspection process. However, the BTS mechanism has no filter itself, incurring in the capture of the `CALL` and `RET` instructions inside libraries in a given process scope. Following code inside libraries might be useful in some situations, but not always. So, we provide the ability to skip these instructions using a filter in the client. This selection may be understood as debugging's `Step Into` and `Step Over` navigation.

Step Into. Figure 4 shows an excerpt from the Step-Into CG from the example code, presenting the analysis of `printf` function internals. After the binary under analysis calls the `printf` entry point, we can find `calls` to internal functions responsible for the `printf` behavior—`locks`, for instance—which assures I/O ordering, since `printf` is a non-reentrant function.

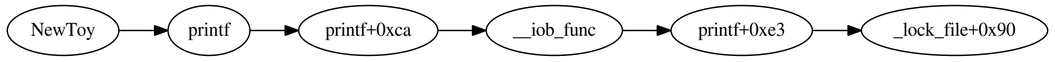


Fig. 4: Step-Into call graph, all intermediate calls represented.

[18]https://sites.google.com/site/branchmonitoringproject/

Step Over. Figure 5 shows the Step-Over CG from the same code in which only the called functions appear. The presented excerpt covers the `return` from the initial `scanf` function at the `0x3f` offset to our binary (`NewToy`) and then the `call` to the entry point of the `printf` function, which will print the read value. Internal aspects of both functions are omitted in this mode.

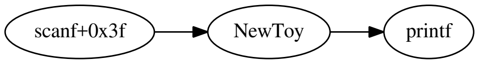
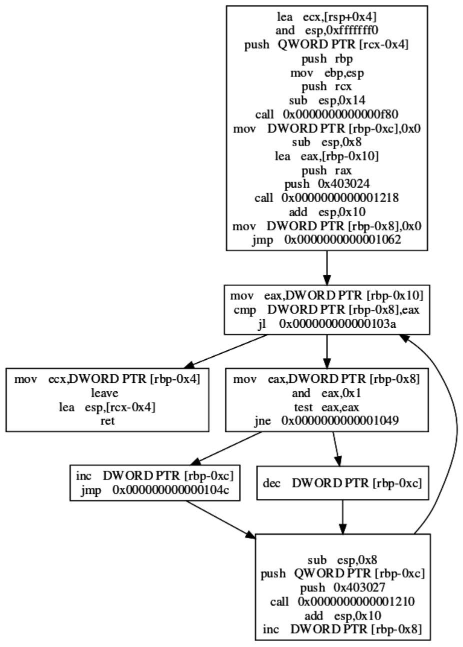


Fig. 5: Step-Over call graph, only CALL/RET represented.

5.1.2. Control Flow Graph. CFG is the most granular inspection view possible of a code at the instruction level. By inspecting it, one can perform taint analysis [Schwartz et al. 2010] or identify malicious payloads [Newsome and Song 2005; Yin et al. 2007]. In order to rebuild the CFG of a given sample, we rely on the disassembly solution previously presented. We apply it repeatedly so that we could retrieve information about each block surrounded by two deviation instructions. Similar to CG's case, the captured data also contains information about library internals. Here, our sole interest is about binary information, so we used the same approach of the Step-Over filtering. For example, the CFG of Figure 6 is result of the following piece of code: `scanf("%d",&n); for(i=0;i<n;i++){ if(i%2==0) a++; else a-; printf("%d",a)}`.



```
lea   ecx,[rsp+0x4]
and   esp,0xfffffff0
push  QWORD PTR [rcx-0x4]
push  rbp
mov   ebp,esp
push  rcx
sub   esp,0x14
call  0x0000000000000f80
mov   DWORD PTR [rbp-0xc],0x0
sub   esp,0x8
lea   eax,[rbp-0x10]
push  rax
push  0x403024
call  0x0000000000001218
add   esp,0x10
mov   DWORD PTR [rbp-0x8],0x0
jmp   0x0000000000001062
```

```
mov   eax,DWORD PTR [rbp-0x10]
cmp   DWORD PTR [rbp-0x8],eax
jl    0x000000000000103a
```

```
mov   ecx,DWORD PTR [rbp-0x4]
leave
lea   esp,[rcx-0x4]
ret
```

```
mov   eax,DWORD PTR [rbp-0x8]
and   eax,0x1
test  eax,eax
jne   0x0000000000001049
```

```
inc   DWORD PTR [rbp-0xc]
jmp   0x000000000000104c
```

```
dec   DWORD PTR [rbp-0xc]
```

```
sub   esp,0x8
push  QWORD PTR [rbp-0xc]
push  0x403027
call  0x0000000000001210
add   esp,0x10
inc   DWORD PTR [rbp-0x8]
```

Fig. 6: Reconstructed CFG from the presented example code.

We can observe a match between the presented code and the generated CFG, where the first block is related to set up routines, such as pushing the stack frame. This block leads to a decision block related to the `for` statement. If the iteration reached its limit, the code proceeds to the left block, where the `main` function is finished by the execution of a `ret` instruction. Otherwise, the execution proceeds to another decision block, associated to the `if` statement—we should notice the `and eax,0x1` instruction, associated to the decision calculation. Even values result in the left block being taken (notice the `a++` represented as an `inc`) and odd ones in the right one (notice the `a--` represented as a `dec`). The `call` in the last block is the `printf` invocation. The execution iterates through the `for` backward edge. As the provided branch information is related to the effectively taken deviation, our solution has the advantage of capturing and disassembling real instructions; it does not suffer from alignment tricks (often used for anti-analysis), which is a common problem on static disassembly.

Some may find similarities between the hereby presented approach and the one presented by Paleari in his Fuzztrace tool[19], detailed in a blog post[20]. In addition to having distinct purposes, the solutions present other differences. Paleari rebuilds the CFG based on perf-supplied edges. The presented study case is a heat map of executed blocks, a task which our solution is also able to perform. Since our framework is modular, a heat map policy would require writing a few lines of code. Paleari's solution, however, is more limited since it does not track external function calls. Our solution, instead, is able to track and introspect into these functions. In addition, we are able to choose how deep we dig into these libraries by selecting the step-into and step-over modes. In the step-into mode, the CFG is rebuilt in the same way as in the `viewer` tool from Fuzztrace. In the step-over mode, however, a stack is used to filter out instruction blocks according to the monitored code (internal or library). All details about our CFG reconstruction algorithm are presented in the next sections. Additionally, Fuzztrace only provides instruction addresses as tracing data, which requires performing a static disassembly in order to match addresses and instructions. Conversely, our solution is able to perform online, dynamic code disassembly, providing the executed block as tracing data.

Self-Modifying Code. Many malware samples perform in-memory code changes, also known as Self Modifying Code (SMC) [Xianya et al. 2015; Debray and Patel 2010]. This is a technique often used for packing samples in order to avoid static detection. Our solution is able to handle packed samples since we can monitor their whole behavior, during and after unpacking (intended execution). In case one wants to monitor the code modification itself, the tracer needs to be run using asynchronous I/O since, in order to reduce overhead, we have implemented the presented version using synchronous I/O, which causes delayed code memory reads. As an advantage, the SMC detection can be performed concurrently with the CFG reconstruction, by the same algorithm, as shown in the next section/paragraph.

The CFG-SMC algorithm. In this section, we detail the CFG reconstruction, covering the step-over execution and external function calls. We also show how we can use the same algorithm (Algorithm 1) to perform SMC detection. The algorithm takes as input a list of instruction blocks and the flags which enable/disable the step-over and SMC detection modes. The algorithm iterates over the instruction block list (line 6), updating the current and previous blocks (line 28), adding edges between them when needed (line 26).

In the step-over mode (line 7), library `CALL`s (line 9) will be exit nodes from the graph whereas `RET`s (line 11) will be entering nodes. In this mode, the instructions in between are not considered, thus the `pass` instruction (line 14). As these blocks are passed, the previous node is kept in the `CALL` instruction and later edge-linked to the node coming after the `RET`. Notice that when the step-over mode is enabled, as the

[19]https://github.com/rpaleari/fuzztrace
[20]http://roberto.greyhats.it/2015/02/fast-coverage-analysis-for-binary.html

libraries are pushed into the stack, the library nodes are printed at a distinct CFG level. If necessary, one may instrument the pass step to generate the whole CFG for a given level (the library CFG, for instance). An example of such generation modes is provided below.

Figure 7 shows the step-into case. As the stack is limited to the first level (`binary`), the library internal code, highlighted in the internal bounded box, is inserted as ordinary binary code blocks. Figure 8 shows the step-over case. As the stack changes from `binary` to `library`, these are printed at distinct levels. The two bounded boxes present, respectively, the binary code (and its library call node) and the library code itself. An existing corner case is related to branches whose targets are instructions inside other blocks. In this case (line 19), the existing block has to be split (line 20) in order to keep the CFG's definition (set of non-branch instructions limited by a branch).

The same block traverse algorithm can be used as base for SMC detection. In this case, a shadow memory is used, thus requiring additional memory. When a block node is created (line 15), its memory content is hashed and stored in a shadow block (line 18). Notice that when a block is split (line 20), block hashes need to be updated (line 22). After the point where the current block is defined, we can check whether the current block hash matches its shadow (line 24). In case any difference is found, an SMC code is identified (line 25). The detection routine can be used to immediately return or update the block hash and keep detecting code changes. Notice that in the SMC case, distinct CFG visualization modes should be used, since the dynamic block content makes plotting harder.

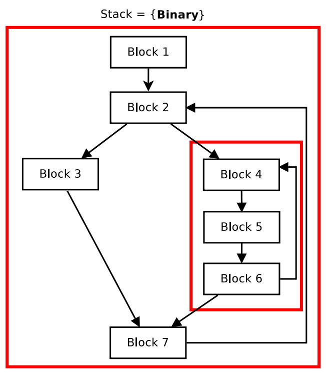


Fig. 7: Step-into CFG.

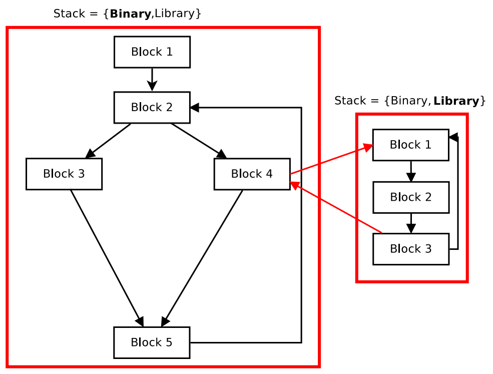


Fig. 8: Step-over CFG.

5.1.3. Modular malware. Many malware use modular approaches to deploy the functions required for infection, such as dropping a file or downloading a payload from the Internet. This way, splitting their maliciousness through many processes actually presents a lower malicious profile. This effectively achieves a lower malware ranking among Anti-Virus tools and is currently a common behavior on modern malware samples. Although our introspection process is able to identify the call to APIs such as `CreateProcess`[21], we are not able to collect the created process PID and thus not able to filter its activities. In order to overcome this limitation, we installed a Process callback[22] which delivers new PIDs to our client to be monitored. This way, the created process is added to the monitored list plus the initial PID, which could be a suspended

[21]https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx
[22]https://msdn.microsoft.com/en-us/library/windows/hardware/ff542860(v=vs.85).aspx

ALGORITHM 1: CFG reconstruction and SMC detection

```
Data: blocks, StepOver, SMC.Detection
Result: CFG, SMC.Detected
1  Stack = {Binary}
2  CFG = ∅
3  Shadow = ∅
4  create_node(block[0],Stack,CFG)
5  previous = block[0]
   /* Iterate over the blocks in an ordered way                                    */
6  for current in blocks[1:n] do
       /* Case filtering is enabled                                                */
7      if StepOver then
           /* Case it's a library. Otherwise, run                                  */
8          if Introspection(current) is LIBRARY then
               /* CALLs are pushed on the stack                                    */
9              if Instruction(current) is CALL then
10                 Stack.push(Library(current))
               /* RETs pop data from the stack                                     */
11             elseif Instruction(current) is RET then
12                 Stack.pop()
               /* Ignore any other internal instruction at the for level           */
13             else
14                 Pass
                   /* Instead of passing, one can generate a CFG for the library   */
       /* Create non-existing nodes in the graph                                   */
15     if not node_exists(current,CFG) then
16         create_node(current,Stack,CFG)
           /* Case SMC.Detection is enabled, compute the block hash the first time */
17         if SMC.Detection then
18             shadow[current]=Hash(Instruction(current))
       /* Case there's a branch to the middle of a previous block                  */
19     elseif not match_first_address(current,CFG) then
           /* split the previous block                                             */
20         current, splitted = split_CFG(current,CFG)
           /* Update block hashes to include the splitted one                      */
21         if SMC.Detection then
22             shadow[splitted]=Hash(Instruction(splitted))
23     if SMC.Detection then
           /* Check current block has the same content than before                 */
24         if shadow[current] is not Hash(Instruction(current)) then
25             SMC.Detected()
       /* add edges                                                                */
26     if not edge_exists(previous,current,Stack,CFG) then
27         create_edge(previous,current,Stack,CFG)
28     previous=current
```

process, as usual on most sandboxes solutions, or even a running process whose PID is known.

5.1.4. Real malware tests. As our tracing tool is built upon our framework, it allows malware analysis in a stealthier way. In order to validate such property, we ran some evasive samples in our environment so as to verify if they executed as expected in a real, victim machine. The samples choice was based on static analysis tools that identify Anti-VM techniques. We selected four samples[23] said to employ QEMU tricks according to PEframe (https://github.com/guelfoweb/peframe) and executed them in our proposed framework and in our QEMU-based internal sandbox solution [Botacin et al. 2017]. All of them evaded the execution in the QEMU-based sandbox, not providing any useful behavior to be analyzed. However, they executed normally under our proposed malware tracer's monitoring.

5.2. Debugger Project and Implementation

The malware tracer allows us to understand a great deal of a sample's behavior through its execution, but it is not able to suspend the execution at an arbitrary point in order to provide a deeper introspection view. This could be a useful approach for detecting bugs or complex constructions, especially those with stealth attack intentions. Extending our framework to provide such facility is a straightforward path.

Goals achievement. In order to achieve the **small step execution** goal, we rely on the BTS mechanism. Although it does not allow step-by-step execution, it provides sequential block-by-block granularity which, with the help of a block disassembler, brings basically the same functionality. The **breakpoint information** goal is achieved by relying on introspection during interrupts. Finally, the **context inspection** goal is achieved by using system APIs. The data consistency is assured due to the raised interrupt which precedes API calls.

Debugger working flow. As we need to suspend the process execution to inspect it, the strategy here is different from the Tracer's. In addition, the process suspension must proceed as soon as an interrupt occurs; to accomplish this, we made use of the inverted I/O call. The debugger working flow is as follows: (i) at a given moment, the processor fetches a deviation instruction whose source and target addresses are stored by the BTS mechanism; (ii) an interrupt is then raised since we have defined a 1-threshold—at this point, the process under analysis is active, but interrupted; (iii) the ISR routine releases the cached I/O in order to alert the user-mode client, which receives the alert, suspends the process execution and finishes the I/O routine; (iv) when the ISR receives the I/O completion signal, the interrupt is released and the process is now in suspended state; (v) then, all introspection and context retrieval processes take place; (vi) when the process is resumed in the client, the whole debugging process is restarted.

Debugger resources. One of the most important resources in a debugger is its inspection capabilities. Our solution presents the following ones:

— **Process management**: our solution is able to create a new suspended process to be inspected or to attach to an existing one.
— **State inspection**: our solution is able to identify function execution, loaded libraries and to read context registers.
— **Step execution**: our solution is able to perform branch step execution at the block, function and library levels.
— **Integration**: our solution can be integrated to other debugging tools, such as GDB.

5.2.1. Debugger client implementation. Although the client was built upon our framework, some features were implemented in the client itself. The details are given below.

Process management. Processes are created using the `CreateProcess` API. Processes need to be at the suspend state in order to be inspected consistently. There-

[23]Samples MD5: f03c0df1f046197019e12f3b41ad8fb2, 2b647bdf374a2d047561212c603f54ea, 7a4b29df077d16c1c186f57403a94356, 340573dd85cf72cdce68c9ddf7abcce6

fore, new processes are created using the `CREATE_SUSPENDED` flag whereas existing ones should be suspended by calling a specific API. There are three known methods for suspending a process: (i) enumerating all threads for a given process and calling `SuspendThread`[24] to each one, which may lead to a deadlock due to thread desynchronization; (ii) calling `DebugActiveProcess`[25], which is detectable by `IsDebuggerPresent`; and iii) using the undocumented API `NtSuspendProcess`, which was used in our solution.

Context values. Context values are obtained by using system APIs. We rely on our framework to perform introspection and disassembly. In addition, register values are retrieved by using the `GetThreadContext`[26] API.

GDB integration. Although we have our own interface to our debugging solution, we opted to integrate it with GDB in order to make use of its extensions and facilities. The integration is done using a `stub`, a small protocol that transfers data from our back-end to GDB. We based our implementation on seaborn's[27] efforts, porting it to Windows. The use of our solution with GDB allows an analyst to inspect Windows systems from distinct platforms and/or over the Internet. The current GDB stub implementation allows for `step` and `info register` commands.

5.2.2. Validation test. To evaluate our approach's stealthiness, we have implemented some *tricks*. Our goal is not to provide an exhaustive list of anti-analysis tricks, but to demonstrate that practical aspects match theoretical ones we have been drawing along this text. We have evaluated the following anti-debugging tricks: `IsDebuggerPresent`, `CheckRemoteDebuggerPresent`, and `OutputDebugString`. None of these tricks were able to detect our solution. For more details, see Appendix G.

We also tested our solution in a real scenario, by inspecting an application protected with an unknown trick. We inspected the `Uplay`[28] binary, a game-launcher, since games are usually protected [Woo and Kim 2012]. The application refused to run under an ordinary debugger but ran under our solution. We were able to perform branch-by-branch execution and read memory contents. For more information, see Appendix H.

5.3. ROP Detector

Given our solution is based on a mechanism that provides branch data, addressing the ROP problem is an immediate follow-up. Indeed, other authors have already leveraged branch monitors for such purposes, such as KBouncer, ROPecker and CFIMon, tools that were presented in Section 3. These approaches, however, are not based on a general framework, as proposed here. Our framework allows inspecting applications with no code injection while solutions like KBouncer require hooking APIs for each process aimed to be monitored. Although such injection requirement does not impose a working restriction for these tools, it restricts the usage scenario. On a general way, such protections are suitable for known vulnerable, unpatched applications in which the ROP protector can be injected. On a broader scenario, where no particular application should be protected but the whole system instead, such injection must occur on all running process. On this scenario, our injection-free, system-wide monitoring approach is a more suitable candidate. In addition, KBouncer and ROPecker make use of a limited number of LBR entries whereas we can use unlimited memory space as we rely on BTS instead.

[24]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms686345(v=vs.85).aspx
[25]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679295(v=vs.85).aspx
[26]https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679362(v=vs.85).aspx
[27]github.com/mseaborn/gdb-debug-stub
[28]www.uplay.com

Implementing ROP policies on our framework is a straightforward task, since it provides us all required information (branch data) and capabilities (process identification, introspection and disassembling). The user-land client can store data in its memory and make use of libraries and data structures, therefore reducing implementation efforts.

In order to assure our approach's correctness, we opted for not developing any new ROP heuristic. Instead, we relied on verified ones. More specifically, we implemented the same two methods used by KBouncer, `CALL-RET matching` and `gadget length`. The `CALL-RET policy` detects a ROP attack by enforcing that each `RET` must be preceded by a `CALL` instruction. Since ROP attacks are based on `RET` instruction chaining, they can be detected.

The gadget length policy is based on the principle that ROP gadgets are usually smaller than legitimate ones. This policy defines a window of the last executed gadgets and their lengths, triggering the detection if a specified number of small gadgets occur. In our solution, we defined the same limits as KBouncer's. When any of the previous policies are violated, an alert is raised. For more details, see Appendix F.

To evaluate our ROP detector, we executed some exploits against vulnerable applications, verifying whether the detection heuristics were triggered or not. The exploits [Son 2011; Knaps 2015] were successful on exploiting the target, being detected by the `CALL-RET` policy. The exploit [Ahrens 2014] crashed during its execution, thus not activating such policy. However, the `Gadget-size` policy was activated instead, due to its small gadgets.

To provide a more qualitative view on ROP detection, we present some more details about Son's exploit. Its execution triggered the gadget length policy; a snippet of the branch window is: first branch target is `0x7c346c0a`; the execution leaves the block at `0x7c346c0b` and reaches `0x7c37a140`; the execution leaves the block at `0x7c37a141`. The instruction disassembly of this code region, from the *MSVCR71.dll 7.10.3052.4 - 32bits* library, is presented in Listing 2.

Listing 2: Static disassembly of the `MSVCR71.dll` library.

```
7c346c08:        f2 0f 58 c3              addsd  %xmm3,%xmm0
7c346c0c:        66 0f 13 44 24 04        movlpd %xmm0,0x4(%esp)
```

The static disassembly provides aligned words. The exploit, however, makes use of an unaligned one, as indicated by the branch to `0x7c346c0a`. If we look to the dynamic disassembly of the corresponding bytes (\x58\xc3), as shown in Listing 3, we identify the actual executed ROP gadget. As expected, our solution detects even unaligned branches.

Listing 3: Dynamic disassembly of the `MSVC71.dll` executed code.

```
0x7c346c0a (byte=0x58)   pop    rax
0x7c346c0b (byte=0xc3)   ret
```

5.4. Anti-Analysis tricks detection

We have implemented a static detector that matches the executed code blocks. Using this detector, we were able to detect the following tricks: `Fake Conditional, Control Flow Change, Hook Detection`, and `Hardware Debugger detection` (see Appendix I).

5.5. Execution deviation detection at branch-level

Despite identifying the use of known anti-analysis tricks through a pattern matching procedure, as previously presented, we can also apply our solution for dynamic trick identification. When a trick leads to an evasion, a branch is taken in order to not execute the malicious payload. If this happens on the emulator but not on the bare metal setup, we can identify the divergence point by comparing the traces. The branch block which has led to the divergence point may present an anti-analysis trick.

This idea is exemplified in Figure 9. The block `0x2` presents an anti-analysis trick. When running on bare metal, the execution proceeds to the `0x3` block whereas it proceeds to the `0x4` block when running on the emulator. For the sake of simplicity, we assume the execution flow will consolidate onto a single block (`0x5`). It can be understood as a common cleanup routine, for instance.

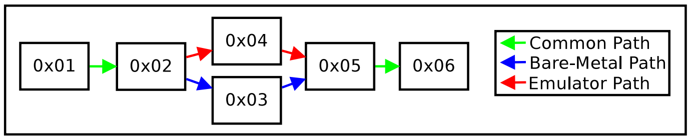


Fig. 9: Example of a flow divergence between the code running on bare metal and on the emulated monitor.

If we assume this property and consider the execution on bare metal as the groundtruth, we can implement an algorithm for deviation detection, as presented in Algorithm 2. The first step consists of discarding the base image addresses and considering just the offsets (line 3), due to ASLR. This way, the traces are now comparable and since they will differ, the second step consists of finding an alignment (line 4) using a global sequence alignment algorithm[29]. After that, given the expected CFG structure we have defined, the aligned traces will be aligned in the beginning (blocks `0x1 and 0x2`) and in the end (blocks `0x5 and 0x6`). This way, the blocks in between are the deviating ones and the last aligned block is the one possibly having the anti-analysis trick. The algorithm proceeds by traversing the blocks, taking the bare metal trace as reference (line 5). When the blocks are aligned (line 6), they are just printed (line 7). When they are not aligned (line 8), we iterate one of the sides (line 11) in order to achieve another aligned block (block `0x5`) (more details on Appendix J).

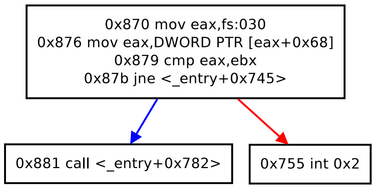
We have evaluated the proposed approach on real samples, by comparing their executions under our solution and others. The compared solutions were our branch monitor solution built upon Intel PIN, presented in Section 4.3, and OllyDbg[30]. We manually inspected the diverging points in order to find possible anti-analysis tricks. Figure 10 illustrates a divergence case due to an anti-analysis trick—checking for the `NtGlobalFlag` (offset `0x68`) in the PEB structure (`fs:0x30` offset)—in the instruction block right before the diverging branch. Some cases, however, are just false positives, since we could not identify any anti-analysis trick. As an example, Figure 11 shows a diverging behavior related to some kind of random decision (`<rand>` call). As future work, an automated decision mechanism may be implemented.

We analyzed 15 random samples that presented divergent-like behavior in our dataset. As a general result, some anti-analysis tricks were found (see Appendix K). The remaining samples turned out to be false positives. The proposed approach does present limitations, such as the ones related to the CFG format. However, our main

[29]Python's alignment library
[30]http://www.ollydbg.de/

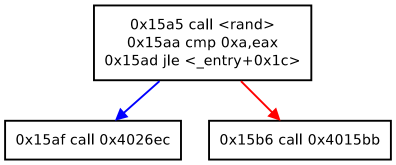
ALGORITHM 2: Flow deviation detection

```
Data: Bare Metal trace (BM), Emulated trace (E)
Result: Deviated Block
i = 0
j = 0
/* Calculate branch offsets                                    */
Bare Metal,Emulated = get_offsets(BM,E) /* Align traces        */
seq1, seq2 = align(Bare Metal,Emulated)
/* Bare Metal trace is reference                               */
while seq1[i] not EOF do
    if seq1[i]==seq2[j] then
        emit_aligned(seq1[i],seq2[j])
    else
        emit_unaligned(seq1[i])
        i++
        while seq1[i]!=seq2[j] do
            emit_unaligned(seq2[j])
            j++
```



```
0x870 mov eax,fs:030
0x876 mov eax,DWORD PTR [eax+0x68]
0x879 cmp eax,ebx
0x87b jne <_entry+0x745>
0x881 call <_entry+0x782>
0x755 int 0x2
```

Fig. 10: True divergence.



```
0x15a5 call <rand>
0x15aa cmp 0xa,eax
0x15ad jle <_entry+0x1c>
0x15af call 0x4026ec
0x15b6 call 0x4015bb
```

Fig. 11: False divergence.

goal is not to fully develop a tool for behavior divergence detection but to suggest how this kind of solution can benefit from using a branch monitoring-assisted solution—as execution deviation happens through branches, monitoring them is a natural way to identify deviating behavior. As an initial approach, our solution could benefit from other solutions as well as to be used to improve other solutions which already addressed the deviating behavior problem, such as MalGene [Kirat and Vigna 2015] and Differential Slicing [Johnson et al. 2011].

6. DISCUSSION, LIMITATIONS, AND FUTURE WORK

In this section, we provide a general overview of our contributions, current limitations and open opportunities on branch monitoring development.

Framework advances. The proposed solution differs from previous work by not only looking at specific branch data, but also proposing a complete framework to handle this data. Unlike previous work, our solution makes use of the BTS mechanism instead of the LBR one, which allows us new constructs, used to develop a complete analysis framework. This framework is characterized by not requiring any code injection and as such relies on a less intrusive approach than other monitoring tools. Our solution is a lightweight alternative to the state-of-the-art ones, since it requires less development efforts—no BIOS rewriting or hypervisor implementation is required—and presents a smaller overhead—only the monitored core is interrupted and most actions can be offloaded to other cores in a current multicore system or processed offline.

Our implementation does not apply any system patch, being able to run on modern OSs, even if it has KPP[31], for instance.

Bare metal and stealthiness. Requirements for increased stealthiness are fulfilled by using a physical machine, a way that might look like sufficient. This stealthiness is accomplished by not introducing side-effects, a feature which is provided by bare metal, as well as by **not** performing code injection/interference. The second requirement comes from the fact that data collection mechanisms usually require interposing binary calls, a task often performed using hooks or debug attachment. Thus, there are many reported research about anti-hooking and hooking detection [CloudBurst 2016; Roccia 2016; Chailytko and Skuratovich 2016] as well as debugger detection [Branco et al. 2012; Barbosa and Branco 2014]. In this sense, we have presented a real example of an interference detection occurrence. The `Uplay` executable refused to run under a debugger even on a bare metal machine. In turn, it executed under our solution, since we addressed the *non-injection / non-interference* requirement.

Solutions Comparison. Our malware tracer can be directly compared to public available and state-of-the-art sandbox solutions. When evaluated against solutions like `Cuckoo` and `CWSandbox`, for instance, our solution is more transparent, since no code injection is performed (these solutions rely on DLL injection for API hooking)—processor data is used instead—and no virtual machine is used (hypervisor side effects are often used as analysis environment indicators by evasive samples), since our solution is bare metal based. In this sense, ours is closer to the HVM-based ones, such as `Ether` and `MAVMM`, presented in related work. When compared to these solutions, ours presents the same level of stealthiness for user land threats, given that in all approaches the malware code is run on a real processor. Unlike such solutions, our approach is not able to handle kernel malware. This limitation, however, is due to the fact that we used a kernel driver to implement our solution and as such we must assure kernel integrity. This implementation choice, however, gives us advantages when compared to competing solutions: 1) Developing a kernel driver requires less development effort than developing a whole hypervisor, which makes our solution simpler to be implemented; 2) Recompiling a kernel driver is much more portable than re-instrumenting hypervisors, making our solution much more accessible; 3) Trapping only branch deviations at kernel level is less costly than trapping each instruction at hypervisor level, contributing to a much smaller overhead.

Our solution might also be compared to other approaches, such as disassemblers, like `Capstone` (used in our framework), Plasma[32] and `Udis86`[33]. These solutions are not analyzers by themselves, rather mere translators of given byte sequences into instruction opcodes. More importantly, the instruction byte sequence data acquisition procedure comes first. Although such solutions can be directly applied to original binaries (a naive static approach), they are vulnerable to anti-disassembly techniques, used by malicious samples to evade analysis [Branco et al. 2012]. Conversely, our dynamic instruction address collection solution, by relying on processor branch data, is able to provide such disassembly tools over unpacked, ready-to-run code, rendering ineffective most anti-disassembly techniques, such as instruction misalignment.

A commercial disassembly solution which can also be compared in some way to our solution is `IDA Pro`[34]. IDA disassembler performs static code disassembling and also allows for CFG and CG reconstruction. In order to tackle anti-analysis techniques, it relies on dynamic emulation of statically unsolvable pieces of code, therefore mitigat-

[31]Kernel Patch Protection
[32]https://github.com/plasma-disassembler/plasma
[33]http://udis86.sourceforge.net/
[34]https://www.hex-rays.com/products/ida/index.shtml

ing some of them. In fact, many approaches tried to solve the anti-analysis problem by relying on rules to defeat known anti-analysis tricks, such as the ones in Cobra [Vasudevan and Yerraballi 2006a] and Vampire [Vasudevan and Yerraballi 2005]. The major drawback of such approaches is that as soon a new anti-analysis trick is discovered, the software has to be updated and/or recompiled. Bare metal-based solutions like ours, however, are able to handle such new tricks naturally, since the code is executed on a real processor. Nonetheless, running huge amounts of samples on real machines does not scale well. Besides being a disassembler, IDA Pro also presents a complete debugger, whose frontend can be attached to GDB, VMWare, QEMU, BOCHS and others. In this sense, our framework could be extended to provide branch data to the IDA frontend the same way as the aforementioned tools do.

As for the proposed debugger, our solution can be directly compared to the HVM-based HyperDbg and the SMM-based MALT, cited in Sec. 3. Our solution presents the same functionalities of such systems, such as register and memory inspection. The most notable difference is that our solution operates at the branch level, due to the branch monitor inherent working characteristic, whereas the other ones operate at the instruction level. Despite not being able to stop at every instruction, only at every block, our debugger is able to reconstruct every executed instruction sequence by making use of the same introspection procedure used for CFG reconstruction. Just like in the case of the tracer, our debugger is also restricted to the userland level, contrary to other solutions like HyperDbg that are able to analyze at even the kernel land. As previously discussed, this implementation choice gives us many advantages when compared to such solutions. The same discussion is also valid for MALT, as handling code at BIOS level is more expensive than using a kernel driver. When compared to the popular solution GDB, ours is better suited to handle anti-debugging software, since it does not rely on ptrace and also provides the same user friendliness, since it is integrated to the GDB frontend.

Finally, our ROP detector is directly comparable to KBouncer and ROPecker, since the same detection heuristics were implemented. The most significant difference is the way they are implemented, making our usage scenario broader in two ways:

(1) Since we rely on the BTS mechanism instead of the LBR one, we are able to handle larger ROP chains—the BTS mechanism relies on O.S. memory page storage, which is theoretically unlimited, whereas the LBR one is limited to the number of MSR registers present in the processor. Our Haswell processor presents 16 of such registers, which would limit detection to a 16-gadget-length ROP exploit at most;
(2) Our approach does not require code injection, allowing us to monitor the whole system at a time; competing solutions require injecting DLLs on each specific code one intends to monitor. It allows us to monitor the whole system without knowing in advance that a specific application is vulnerable.

Implementation limitations. The main limitation of our solution is the process context inspection mechanism—notably the memory reading mechanism—which is implemented as a userland component, making it less protected from subversion than kernel components. We considered this project decision as acceptable since we are developing a proof of concept application. If more protection is required, these mechanisms can be moved to kernel without significant side effects, apart from the development effort. Another limitation of our solution is related to the ROP scenario. Although we are able to detect its occurrence, we currently cannot block it, since no active component is injected into the process. An external blocking procedure should be implemented if the user is concerned about it. This task is eased due to the fact that our framework is constructed in a modular and independent way, allowing such kind of extensions. Our solution does not handle some code constructions, such as the use of `PUSH+RET` as a replacement for a CALL instruction. This is a frequent assumption with many monitors due to implementation constraints, although no theoretical limitation is observed. We also targeted only single-core threats, since they are the most frequently observed ones. The monitoring platform, however, does not present any limitation to work on a multi-core scenario. In this case, the framework should be extended to work with multiple sources of data, since a malicious action could be spread through many different cores in order to avoid an ordinary pattern matching process. Currently, to prevent a process from migrating to a different core than the one where the mechanism was enabled into (which would break data capture), we attach the process to a specific core by setting processor affinities, an O.S. functionality[35].

The BTS mechanism is configured to capture data only in the userland ring—despite being able to collect data at the kernel level—since we are targeting only userland processes in our threat model. Targeting kernel threats would require a more privileged ring in order to provide the required isolation for the data collection mechanism. This choice leads us to lose execution control when a syscall is invoked, which is not a problem for tracing binaries that only call libraries, but is otherwise a problem when tracing libraries that do perform such calls[36].

Introspection limitations. A sample which employs an external or static library may bypass our introspection procedure, since function names will not be recovered in this case. The execution of these libraries, however, will still be logged by the BTS monitor, allowing post-analysis by a human analyst. Another corner case is about function arguments. As BTS provides only instruction addresses, we are only able to directly get function calls, not their arguments. This is not a limitation *per se*, since some solutions, such as some malware variant detection tools [Zhong et al. 2012], rely on function call structures. Solutions which require function arguments to enrich their usefulness may instrument the function calls indicated by our solution, such as in the modular malware case presented in Section 5.1.3. Notice that, in this case, there will be an overhead penalty according to the added instrumentation mechanism.

Malware Analysis Limitations. Our malware analysis solution suffers from the same limitations others do regarding stimulation, which is directly related to the reached code coverage. In order to overcome such limitation, user interactions can be simulated by using AutoIT scripts [37] or similar ones. However, in the scope of this work, we are concerned about reaching code hidden by anti-analysis techniques, for which stealthier solutions like this play a crucial role. Our solution, as a sandbox, is also subject to fingerprinting, an open problem for all monitoring systems, thus outside the scope of this work.

Sandbox Restore. As our approach is bare-metal based, we have to restore the system to a clean state after running a malicious sample. In virtualized environments, it is usually done by reverting to a VM snapshot. On a bare metal system, as automatic snapshots are not available, it has to be manually done. As a way of automating the task, PXE boot or LVM volumes may be used.

Evasion Scenarios. Every new proposed solution will be targeted by attackers in order to defeat it and so keep their stealthiness. Our solution, as is, relies on PID tracking for process filtering, a feature we believe is the most probable target for attackers: by faking a PID, a malware could make the analysis produce no result. As a countermeasure, we could filter actions not by PID but by the address itself, since each process is mapped to a unique memory region. This change is straightforward

[35]https://msdn.microsoft.com/en-us/library/windows/desktop/ms686223(v=vs.85).aspx
[36]Distinctly from Linux, Windows applications do not call the O.S. directly, rather they use O.S. libraries.
[37]https://www.autoitscript.com/site/autoit/

in our solution since the address is exactly the data provided by the BTS mechanism. Another implementation possibility is to rely on CR3-based introspection techniques.

Solution Portability. An important consideration regarding the proposed solution is about its application on other systems and architectures. In the first case, the system is portable since its main component is a hardware-resource, so we have to port only the introspection procedure to the target O.S. Porting our system from Windows to Linux, for instance, would require only changing DLL imports to a system call table when interpreting target addresses. In fact, this port is a current work-in-progress. Regarding the architectural support, our solution depends on a branch monitor mechanism able to provide source and target address data. Despite relying on Intel's facilities, we are aware that similar mechanisms are also present in the AMD and ARM platforms. Further investigation is required to develop a port to these architectures.

Portability and Linux. The Linux kernel provides an interface for accessing many performance counters, including branch monitors. These interfaces are used by some tools, such as the `perf` profiler. The simple, direct use of these interfaces, however, does not answer many of the stated questions in this work, such as introspection or code reconstruction. Besides, these interfaces present some limitations, such as being disabled in the kernel [Soffa et al. 2011]. This way, it is fully justifiable to develop/port a framework like the one hereby presented to the Linux environment.

ROP Scenario. This solution is not intended to be the definitive one, since new ways of constructing gadgets have been constantly presented [Schuster et al. 2014] and new deviation attacks have been developed, such as Jump Oriented Programming (JOP) [Bletsch et al. 2011b] and Loop Oriented Programming (LOP) [Lan et al. 2015]. In addition, ROP can be seen as only the tip of an iceberg in the code-reuse attack scenario, since other constructions, like for instance the weird machines ([Vanegue 2014; Bangert et al. 2013; Shapiro et al. 2013]), may arise in the near future as practical and widespread attacks. However, monitoring ROP will still be a required task for security purposes, such as countermeasure development or forensic procedures. Our solution presented advances by monitoring the whole system without requiring injection and providing a framework which allows monitoring unorthodox constructs. Therefore, building tools relying on previous assured characteristics, such as data collection with minimal fingerprint, is now an easier task.

Overhead. The branch monitor mechanism theoretically presents zero overhead, since it is a hardware component that runs independently from the main CPU processing. Some work, however, suggested some considerable impact [Soffa et al. 2011]. We confirmed a negligible overhead by measuring the activation overhead[38]—less than 1%—for both LBR and BTS. Despite the low impact of this stage, data collection and analysis add overhead to the system, since an interrupt is raised and memory access and I/O are performed. This overhead is application dependent: a delayed collection for malware tracing adds less overhead than the real-time ROP detection approach. This way, we opt to split the overhead by tasks.

To do so, we developed a tiny program—compiled with no optimizations—which takes a million branches and measured its execution on a dedicated core with and without the running framework, using synchronous I/O. Data collection in the client, including interrupt and I/O, adds a 14% overhead; the introspection process adds another 26% overhead; the total overhead when both are combined may go up to 43%. When the introspection handling was moved to the same core as the monitored application, the overhead grew up to 75% on the small test program. These results are still smaller than the related tools Ether and MAVMM, for example, which present, in some cases, overheads of around 72% and 100%, respectively. Another evaluated scenario is

[38]With no data handling.

changing the delayed execution collection to a real time monitor using asynchronous I/O. On these tests, we measured overheads of 100%. As a comparison, the PIN tool used for validation purposes presented overheads of 400% in the same scenario. This 4x higher overhead matches Paleari's findings on his Fuzztrace solution.

Speeding the monitoring up may be done through moving the analysis to another core/processor whenever possible, such as in related approaches [Quinn 2012]. As for disassembly, we can build a disassembly database from relevant and trustable portions of code, such as system libraries, avoiding the cost of a dynamic disassembly. This approach would be similar to what ROPecker applies to its gadgets. We also evaluated the impact of our solution in real scenarios. Benchmark results are presented in Appendix L.

Apart from these evaluations, we performed some tests to compare BTS and LBR in distinct scenarios. Firstly, we evaluated the impact of the data collection procedure on the test program. As mentioned, the BTS use imposed a 14% overhead. When using a high-rate[39], software interrupt-based polling approach for LBR collected data, the overhead grows to 26%. These results match CERN's results [Bitzes and Nowak 2014], which reported overheads from 16% to 25%, depending on applications. We tried to vary the polling interval time from 1ms to 1000ms. The overhead started to decrease after the 200ms threshold, possibly causing data loss, thus showing our correct choice for BTS instead. We also performed the same experiment of threshold variation for the BTS hardware interrupt threshold. We measured a decreased performance impact only after a 50-instruction threshold, which shows the ISR handling itself as the most performance-expensive event. We also tried to evaluate the instruction filtering effect provided by the LBR mechanism. We noticed an overhead decrease of 6% when handling only `CALL` data compared to the general case. The result was 3% when handling only `JMPs`. This impact, however, is application/system-dependent, since it is impacted by the frequency of such instructions in the executed code. In order to better demonstrate this point, we implemented the basic handling mechanisms on Linux, so that we could compare both OSs. The Linux base performance value is 4% lower than on Windows, which reflects system differences; the same result is seen when handling the BTS interrupt. The Linux overhead is 6% lower than on Windows. This way, we conclude that the performance impact should be evaluated in each usage scenario by considering distinct OSs, applications and architectures.

Hardware-Assisted Approaches for ROP-CFI The main advantage of the proposed branch-monitor-assisted approach when compared to software-dependent solutions is that no recompilation or binary-rewriting is required. However, if it is not the usage case, other hardware-assisted approaches are alternative candidates. HAFIX [Davi et al. 2015] extends the instruction set to add CFI instructions which implement the same CALL-RET policy here presented. As no instruction-level monitoring is required, the overhead is significantly smaller. However, the usage of such instructions depends on (re)compiling the code with the newly added CFI instructions. The official proposal to extend the x86 architecture to implement a CFI policy was presented by Intel in its Control Flow Enforcement technology [Intel 2016], whose CFI policy is implemented through a shadow stack, a distinct yet related approach to the ones previously presented.

6.1. Suggestions for Branch Monitoring Improvement

The BTS and LBR mechanisms were originally developed aiming at profiling issues. However, as researchers tried to turn them onto a security-oriented monitoring plat-

[39]Using Windows kernel timers

form, some resource gaps are apparent. In this section, we pinpoint some missing features we wish were present on the monitoring platform – focusing on the BTS.

Although BTS supports some kind of filtering, such as userland/kernel capture, it does not support all the filters of the LBR mode, such as the branch-type-based one. The implementation of such feature in BTS would allow for more granular policy implementations, such as those which relies on indirect branches (JOP, for instance).

Despite using the same interrupt vector and O.S. pages, the PEBS mechanism supplies richer context information than the BTS one. For instance, PEBS is able to provide register value information on its data units. If such data were provided also on BTS units, solutions like our debugger proposal would be easier to implement, since data acquisition would be straightforward.

Having more context data could also allow for fast data processing. If some process isolation information were available, the introspection procedure would be simplified. The concept of an O.S. process is not defined at the processor level, but having register information, such as the CR3[40], unique for each process, would ease the filtering task.

We are aware that many of the proposed features might be unfeasible or hard to implement in a mechanism originally not intended for such tasks, due to either design constraints or increased costs. As such, developing an independent monitoring platform which works in a similar way to BTS and LBR might be a better choice for processor improvement. This kind of proposal tends to look more attractive as computer systems get more complex to instrument. We see Intel's `Processor Tracer` [R. 2013] as a first step toward such direction.

6.2. Future Work

An immediate extension of our framework is the implementation of new policies and monitors, since it provides complete support for such developments. In addition, the Intel platform brings other opportunities, such as extending our framework to work with PEBS data, which would allow one to develop distinct policies in the userland client. These policies include, for instance, malware detection through side effect measurements.

7. CONCLUSION

In this paper, we have introduced an extensible framework for software analysis less prone to fingerprinting, which is based on performance monitoring hardware features. We have shown how the framework can be applied to convey dynamic malware analysis, debugging facilities and a ROP detector tool. Our work is intended to be a stealth, lightweight solution compared to other state-of-the-art developments.

REFERENCES

Julien Ahrens. 2014. Easy File Management Web Server 5.3 - UserID Remote Buffer Overflow (ROP). https://www.exploit-db.com/exploits/33610/. (2014). Access Date: 2017.

Y. Akao and T. Yamauchi. 2015. Proposal of Kernel Rootkits Detection Method by Monitoring Branches Using Hardware Features. In *2015 IIAI 4th Intl. Congr. Adv. Applied Informatics*. IEEE, 721–722.

Erdem Aktas and Kanad Ghose. 2013. Run-time Control Flow Authentication: An Assessment on Contemporary x86 Platforms. In *Proc. 28th Annual ACM Symp. Applied Computing (SAC '13)*. ACM, 1859–1866.

AMD. 2012. *AMD64 Architecture Programmer's Manual Volume 2*. AMD.

ARM. 2011. *Cortex-A Series Programmer's Guide*. ARM.

Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symp.* 1–16.

[40]page directory base register - PDBR

Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. 2013. The Page-fault Weird Machine: Lessons in Instruction-less Computation. In *Proc. 7th USENIX Conf. Offensive Technologies (WOOT'13)*. 1–13.

Gabriel Negreira Barbosa and Rodrigo Rubira Branco. 2014. Prevalent Characteristics in Modern Malware. http://www.kernelhacking.com/rodrigo/docs/blackhat2014-presentation.pdf. (2014).

Georgios Bitzes and Andrzej Nowak. 2014. The overhead of profiling using PMU hardware counters. https://zenodo.org/record/10800/files/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf. (2014). Access Date: May/2017.

Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011a. Mitigating Code-reuse Attacks with Control-flow Locking. In *Proc. 27th Annual Computer Security Applications Conf. (ACSAC '11)*. ACM, 353–362.

Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011b. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proc. 6th ACM Symp. Inform. Comp. Comm. Security (ASIACCS '11)*. 30–40.

Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. 2017. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques* (27 Feb 2017). DOI:http://dx.doi.org/10.1007/s11416-017-0292-8

Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti-VM Technologies. http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf. (2012).

Capstone. 2016. The Ultimate Disassembly Framework. http://www.capstone-engine.org/. (2016). Access Date: July/2016.

Alexander Chailytko and Stanislav Skuratovich. 2016. VB2016 paper: Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment. https://www.virusbulletin.com/uploads/pdf/magazine/2016/VB2016-Chailytko-Skuratovich.pdf. (2016).

Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. 2009. DROP: Detecting Return-Oriented Programming Malicious Code. In *Proc. 5th Intl. Conf. Information Systems Security (ICISS '09)*. 163–177.

Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, Huijie DENG, and others. 2014. ROPecker: A generic and practical approach for defending against ROP attack. *NDSS Symp. 2014* (2014).

Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. 2015. Practical Domain-specific Debuggers Using the Moldable Debugger Framework. *Comput. Lang. Syst. Struct.* 44, PA (Dec. 2015), 89–113.

CloudBurst. 2016. Reverse Engineering for Malware: Shellcodes and AV/API Hook Evasion. https://www.cloudburstsecurity.com/2016/06/10/reverse-engineering-for-malware-shellcodes-and-avapi-hook-evasion. (2016).

L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. 2015. HAFIX: Hardware-Assisted Flow Integrity eXtension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conf. (DAC)*. ACM/IEEE, San Francisco, CA, USA, 1–6.

Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2009. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks. In *Proc. of the 2009 ACM Workshop on Scalable Trusted Comp. (STC '09)*. ACM, New York, NY, USA, Article -, 6 pages.

S. Debray and J. Patel. 2010. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *2010 17th Working Conf. on Reverse Engineering*. IEEE, Beverly, MA, USA, 131–140.

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proc. 15th ACM Conf. Computer and Comm. Security (CCS '08)*. 51–62.

Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2 (March 2008), 6:1–6:42.

Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. 2010. Dynamic and Transparent Analysis of Commodity Production Systems. In *Proc. IEEE/ACM Intl. Conf. Automated Software Engineering (ASE '10)*. 417–426.

Yuxin Gao, Zexin Lu, and Yuqing Luo. 2014. Survey on malware anti-analysis. In *Intelligent Control and Information Processing (ICICIP), 2014 Fifth Int. Conf. on*. IEEE, Dalian , China, 270–275.

Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symp. (USENIX Security 14)*. USENIX Association, San Diego, CA, 417–432.

Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *ASIACCS 2016, 11th ACM Asia Conf. Computer Comm. Security*.

Groundworkstech. 2016. A Python interface to the GNU Binary File Descriptor (BFD) library. https://github.com/Groundworkstech/pybfd. (2016). Access Date: July/2016.

Claudio Guarnieri. 2013. Cuckoo Sandbox. http://www.cuckoosandbox.org/. (2013).

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'D My Gadgets Go?. In *Proc. 2012 IEEE Symp. Security and Privacy (SP '12)*. 571–585.

A. Ho, S. Hand, and T. Harris. 2004. PDB: pervasive debugging with Xen. In *Grid Comp., 2004. Proc.. Fifth IEEE/ACM Int. Workshop on*. IEEE/ACM, Pittsburgh, USA, 260–265.

intel. 2015. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel.

Intel. 2016. Control-Flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. (2016). Access Date: January/2017.

Jun Jiang, Xiaoqi Jia, Dengguo Feng, Shengzhi Zhang, and Peng Liu. 2011. HyperCrop: A Hypervisor-based Countermeasure for Return Oriented Programming. In *Proc. 13th Intl. Conf. Information and Communications Security (ICICS'11)*. 360–373.

Noah M. Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11)*. IEEE Computer Society, Washington, DC, USA, 347–362. DOI:http://dx.doi.org/10.1109/SP.2011.41

Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 769–780. DOI:http://dx.doi.org/10.1145/2810103.2813642

Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. Barecloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proc. 23rd USENIX Security Symp. (SEC'14)*. 287–301.

Knaps. 2015. Easy File Sharing Web Server 7.2 - Remote Buffer Overflow (SEH) (DEP Bypass with ROP). https://www.exploit-db.com/exploits/38829/. (2015). Access Date: 2017.

S. Kompalli. 2014. Using Existing Hardware Services for Malware Detection. In *Security and Privacy Workshops (SPW), 2014 IEEE*. IEEE, San Jose, CA, 204–208.

S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. 2010. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *2010 19th IEEE Asian Test Symp.* 59–64.

Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. 2015. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *IEEE Trustcom/BigDataSE/ISPA*. 190–197.

Jari-Matti Mäkelä, Ville Leppänen, and Martti Forsell. 2013. Towards a Parallel Debugging Framework for the Massively Multi-threaded, Step-synchronous REPLICA Architecture. In *Proc. 14th Intl. Conf. Computer Systems and Technologies (CompSysTech '13)*. ACM, NY, USA, Article -, 8 pages.

J.A.P. Marpaung, M. Sain, and Hoon-Jae Lee. 2012. Survey on malware evasion techniques: State of the art and challenges. In *IEEE ICACT, 14th Intl. Conf. Advanced Comm. Technology*. 744–749.

A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Computer Security Applications Conf., 2007. ACSAC 2007. Twenty-Third Annual*. ACM, Miami, FL, 421–430.

James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. 12th Annual Network and Distributed System Security Symp. (NDSS'05)*. 1–17.

Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. 2009. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proc. IEEE Annual Computer Security Applications Conf. (ACSAC'09)*. 441–450.

Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries. In *Proc. of the 26th Annual Computer Security Applications Conf. (ACSAC '10)*. ACM, New York, NY, USA, Article -, 10 pages.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proc. of the 2012 IEEE Symp. on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, Article -, 15 pages.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc.22Nd USENIX Security Symp. (SEC'13)*. 447–462.

Rian Quinn. 2012. *Detection of malware via side channel information*. Ph.D. Dissertation. Binghamton Univ.

James R. 2013. Processor Tracing. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. (2013). Access Date: May/2017.

Thomas Roccia. 2016. An Overview of Malware Self-Defense and Protection. https://securingtomorrow.mcafee.com/mcafee-labs/overview-malware-self-defense-protection/. (2016).

Jonathan B. Rosenberg. 1996. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA.

Christian Rossow, Christian J. Dietrich, Christian Kreibich, Chris Grier, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *Proc. 33rd IEEE Symp. Security and Privacy (S&P)* . 65–79.

Daniel Schulz and Frank Mueller. 2000. A Thread-aware Debugger with an Open Interface. In *Proc. 2000 ACM SIGSOFT Intl. Symp. Software Testing and Analysis (ISSTA '00)*. ACM, USA, Article -, 11 pages.

Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. 2014. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Research in Attacks, Intrusions and Defenses: 17th Int. Symp. (RAID 2014)*. Springer, Gothenburg, Sweden, 88–108.

Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. Security and privacy (SP)*. IEEE, IEEE, Berkley, CA, 317–331.

Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. 2013. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *Proc. 7th USENIX Conf. on Offensive Technologies (WOOT'13)*. 11–11.

Ahmad Sharif and Hsien-Hsin S. Lee. 2008. Total Recall: A Debugging Framework for GPUs. In *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH '08)*. 13–20.

Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *23rd USENIX Security Symp. (USENIX Security 14)*. USENIX Association, San Diego, CA, 271–285.

Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software* (1st ed.). No Starch Press, San Francisco, CA, USA.

M. L. Soffa, K. R. Walcott, and J. Mars. 2011. Exploiting hardware advances for software testing and debugging: NIER track. In *33rd Intl. Conf. Software Engineering (ICSE'11)*. 888–891.

Nguyen Hong Son. 2011. ROP chain for Windows 8. http://security.bkav.com/home/-/blogs/rop-chain-for-windows-8/normal. (2011). Access Date: 2017.

J. Vanegue. 2014. The Weird Machines in Proof-Carrying Code. In *Security and Privacy Workshops (SPW), 2014 IEEE*. IEEE, San Jose, CA, 209–213.

Amit Vasudevan and Ramesh Yerraballi. 2005. Stealth Breakpoints. In *Proc. 21st IEEE Annual Computer Security Applications Conf. (ACSAC '05)*. 381–392.

Amit Vasudevan and Ramesh Yerraballi. 2006a. Cobra: Fine-grained Malware Analysis Using Stealth Localized-executions. In *Proc. IEEE Symp. Security and Privacy (SP '06)*. 264–279.

Amit Vasudevan and Ramesh Yerraballi. 2006b. SPiKE: Engineering Malware Analysis Tools Using Unobtrusive Binary-instrumentation. In *Proc. 29th Australasian Comp. Science Conf. (ACSC '06)*. 311–320.

Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of the 2012 ACM Conf. on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, Article -, 12 pages.

C. Willems, T. Holz, and F. Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5 (March-April 2007), 32–39. Issue 2.

Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the Bare Metal: Using Processor Features for Binary Analysis. In *Proc. of the 28th Annual Computer Security Applications Conf. (ACSAC '12)*. ACM, New York, NY, USA, Article -, 10 pages.

Jiyoung Woo and Huy Kang Kim. 2012. Survey and Research Direction on Online Game Security. In *Proc. of the Workshop at SIGGRAPH Asia (WASA '12)*. ACM, New York, NY, USA, Article -, 7 pages.

Yubin Xia, Yutao Liu, H. Chen, and B. Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Intl. Conf. Depend. Systems Networks (DSN 2012)*. 1–12.

M. Xianya, Z. Yi, W. Baosheng, and T. Yong. 2015. A Survey of Software Protection Methods Based on Self-Modifying Code. In *IEEE Intl. Conf. Computational Intelligence and Comm. Networks (CICN)*. 589–593.

Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM conference on Computer and communications security*. ACM, ACM, Miami, Florida, 116–127.

Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. 2011. Security Breaches As PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters. In *Proc. of the Second Asia-Pacific Workshop on Systems (APSys '11)*. ACM, New York, NY, USA, Article 6, 5 pages.

F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *IEEE Symp. Security and Privacy (SP)*. IEEE, San Jose, CA, 55–69.

Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. SPECTRE: A Dependable Introspection Framework via System Management Mode. In *Proc. 43rd Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN '13)*. 1–12.

Y. Zhong, H. Yamaki, and H. Takakura. 2012. A Malware Classification Method Based on Similarity of Function Structure. In *IEEE/IPSJ 12th Intl. Symp. Applications and the Internet*. 256–261.

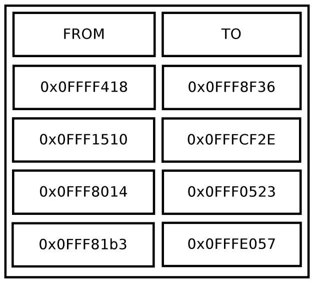
Appendix for the paper: Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging

Marcus Botacin[1], Paulo de Geus[2], André Grégio[3],
(1) University of Campinas
Email: marcus@lasca.ic.unicamp.br
(2) University of Campinas
Email: paulo@lasca.ic.unicamp.br
(3) Federal University of Paraná
Email: gregio@inf.ufpr.br

We present in this appendix additional material which was not included on the main text due to space constraints.

A. THE BRANCH STACK

Branch data is stored on a stacked way, both on LBR and BTS modes. Figure 12, illustrates a branch stack. The `FROM` and to `VALUES` are instruction addresses. The values are stored from the upper entries to the lower. When using LBR, these entries are MSRs whereas these are memory entries when using BTS.



FROM	TO
0x0FFFF418	0x0FFF8F36
0x0FFF1510	0x0FFFCF2E
0x0FFF8014	0x0FFF0523
0x0FFF81b3	0x0FFFE057

Fig. 12: Example of a Branch Stack.

B. ASLR EFFECT

To illustrate the ASLR effect over code images placement, we have checked libraries's addresses after consecutive reboots, as shown in Table I.

Table I: ASLR - Library placement after two consecutive reboots.

Library	NTDLL	KERNEL32	KERNELBASE
Address 1	0xBAF80000	0xB9610000	0xB8190000
Address 2	0x987B0000	0x98670000	0x958C0000

C. FUNCTION OFFSETS

Our solution relies on address introspection in order to provide information on a higher semantic level. Table II shows function offsets for the `NTDLL` library, as an example.

Table II: Function Offsets from `ntdll.dll` library.

Function	***Offset***
NtCreateProcess	0x3691
NtCreateProcessEx	0x30B0
NtCreateProfile	0x36A1
NtCreateProfileEx	0x36B1
NtCreateResourceManager	0x36C1
NtCreateSemaphore	0x36D1
NtCreateSymbolicLinkObject	0x36E1
NtCreateThread	0x30C0
NtCreateThreadEx	0x36F1

D. CALL-RET OPCODES

The `CALL-RET` policy for ROP detection is based on instruction opcodes matching. The bytes representing the CALL and RET instructions are shown respectively in Table III and Table IV.

Table III: `CALL` Opcodes.

Opcode	Mnemonic	Opcode	Mnemonic
0xE8	CALL rel16	0x9A	CALL ptr16:16
0xE8	CALL rel32	0x9A	CALL ptr16:32
0xFF	CALL r/m16	0xFF	CALL m16:16
0xFF	CALL r/m32	0xFF	CALL m16:32

Table IV: `RET` Opcodes.

Opcode	Mnemonic	Opcode	Mnemonic
0xC3	RET	0xC2	RET imm16
0xCB	RET	0xCA	RET imm16

E. PIN VALIDATION

In order to validate our framework, we have implemented the same ideas on Intel PIN, a dynamic binary translator. In the emulated prototype, we considered only branch data and reconstructed instruction blocks by relying on two consecutive branches, as shown in Listing 4.

Listing 4: Instruction Instrumentation on PIN.

```
VOID Instruction(INS ins, VOID *v){
    if(INS_IsBranchOrCall(ins))
        Disasm(last,current)
```

We have run sample programs on both solutions and compared the results for each execution, considering the framework as correct since all of them matched. As an example, Listing 5 and Listing 6 present the execution of a given piece of code under PIN and our solution, respectively. One can verify that the execution of the same block (`0x90` offset)[41] resulted on the same number of disassembled instructions (`0xc96 - 0xc90 = 0x7`).

Listing 5: Sample code running under PIN.

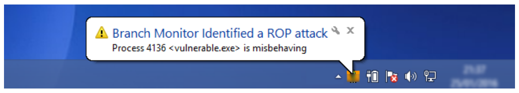
```
From: 0000000077332F89 To: 0x7732ec90 Disasm of 1 instr: call
From: 000000007732EC97 To: 0x7732ecab Disasm of 1 instr: jnz
Disasm of 0x7 bytes from 000000007732EC90: 0x48 0x3b 0xd 0x39 0x8e 0xe 0x0
```

Listing 6: Sample code running under Branch Monitor.

```
Binary Branch.Tester.exe at <0x1ca1> to Binary Branch.Tester.exe at <0x1c90>
Binary Branch.Tester.exe at <0x1c96> to Binary Branch.Tester.exe at <0x1c9a>
should disasm from 7ff6d6ec1c90 to 7ff6d6ec1c96
```

F. RAISING ALERTS FOR ROP ATTACKS

When a ROP attack is detected, an alert is raised, as shown in Figure 13.



Branch Monitor Identified a ROP attack
Process 4136 <vulnerable.exe> is misbehaving

Fig. 13: Alert raised by our solution when an attack is detected.

G. EVALUATING DEBUGGER'S RESISTANCE AGAINST EVASIVE MALWARE

We present in this section the anti-debugging tricks we have used to check our solution's increased stealthiness.

IsDebuggerPresent. It is the default way of checking a debugger's presence on Windows. This code (shown in Listing 7) detected the debugger when running under ordinary debuggers, but not on our system.

Listing 7: Simplest debugger detection code.

```
if(IsDebuggerPresent())
    printf("debugged\n");
else
    printf("NO DBG\n");
```

CheckRemoteDebuggerPresent. A way of checking debugger's presence on a remote host that is also able to detect whether a process is being debugged when attached to itself. The code (shown in Listing 8) did not detect the debugger on our system.

[41] Base addresses are changed on distinct executions

Listing 8: 2nd Simplest debugger detection code.

```
CheckRemoteDebuggerPresent(GetCurrentProcess(),&result);
if(result)
    printf("debugged\n");
else
    printf("NO DBG\n");
```

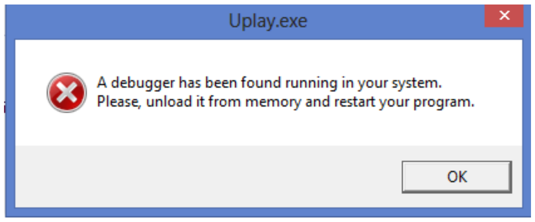
OutputDebugString. This function is the default way of printing a message in the debugger. The resulting `eax` values changes according to whether the debugger is attached or not, thus allowing the debugger presence detection. This code (shown in Listing 9) did not detect the debugger's presence on our system.

Listing 9: 3rd Simplest debugger detection code.

```
OutputDebugStringA(OUTPUT_MSG);
__asm {mov result, eax;}
if(result==DEBUGGED)
        printf("debugged\n");
else
        printf("NO DBG\n");
```

H. INSPECTING A REAL APPLICATION

We show in this section how a real application behaves under an ordinary debugger and under our solution. Figure 14 shows how the binary refuses to run under an ordinary debugger, whereas Figure 15 shows the inspection under our solution.


Uplay.exe

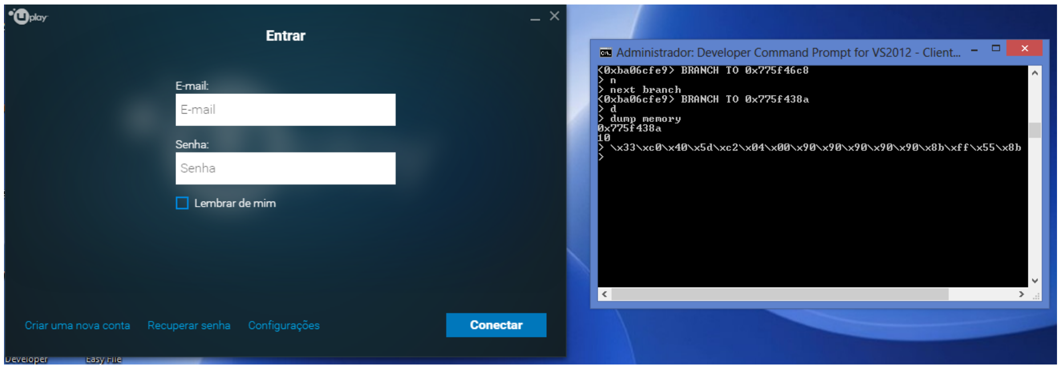
A debugger has been found running in your system.
Please, unload it from memory and restart your program.

Fig. 14: Uplay execution under an ordinary debugger.

I. DETECTING ANTI-ANALYSIS TRICKS

We have made use of our solution to detect some anti-analysis tricks in practice. We describe, below, how the detected tricks work.

Listing 10 presents an identified example of the `Fake Conditional` trick. In order to confuse solutions that follow the executed paths, this trick tries to purposefully trigger the path explosion problem [Krishnamoorthy et al. 2010]. Notice that in practice the branch will always be taken, given the `xor` instruction always yields zero.



```
<0xba06cfe9> BRANCH TO 0x775f46c8
> n
> next branch
<0xba06cfe9> BRANCH TO 0x775f438a
> d
> dump memory
0x775f438a
10
> \x33\xc0\x40\x5d\xc2\x04\x00\x90\x90\x90\x90\x90\x8b\xff\x55\x8b
>
```

Fig. 15: Uplay execution under our solution.

Listing 10: Fake Conditional.

```
0x190 xor eax, eax
0x192 jz 0x19c
```

A variation of this technique consists in changing the unconditional jump to a distinct instruction. Listing 11 shows a trick that changes the control flow by pushing a value to the stack and then returning to it.

Listing 11: Control Flow Change.

```
0x180 push 0x10a
0x185 ret
```

Some samples try to detect the presence of a hook, which may indicate it is under analysis. This is done by checking the presence of the `JMP` instruction (byte `0xe9`). A real example is shown in Listing 12.

Listing 12: Hook Detection

```
0x340 cmp eax,0xe9
0x345 jnz 0x347
```

Some samples perform a similar detection in order to detect the presence of a hardware debugger. The example in Listing 13 shows the presence checking of the debugger register 0 (`0x4`) inside the debugger context struct (`0xc`).

Listing 13: Hardware Debugger Detection

```
0x400 QWORD PTR fs:0x0,rsp
0x409 mov     rax,QWORD PTR [rsp+0xc]
0x40e cmp     rbx,QWORD PTR [rax+0x4]
```

J. DEVIATION DETECTION ALGORITHM OUTPUT

Algorithm 2's execution, using the inputs for the example presented in Figure 9 (`0x1 0x2 0x3 0x5 0x6` and `0x1 0x2 0x4 0x5 0x6`, respectively), resulted in the output presented in Listing 14.

Listing 14: Flow deviation identification by applying the alignment algorithm.

```
0x01 | 0x01
0x02 | 0x02
    / \
0x03 | 0x04
    \ /
0x05 | 0x05
0x06 | 0x06
```

K. DETECTED TRICKS DUE TO DIVERGENT BEHAVIOR

We detail, in Table V, the tricks detected through using our solution.

Table V: Anti-analysis tricks found due to branch-diverged behavior.

# of samples	Trick	Description
2	PUSH-RET	Replacing a `CALL` by a stack-pushed value
2	Fake Conditional	`XOR` itself to trigger branch-related flags
1	NtGlobalFlag	Checking data related to the process heap
1	Hook Detection	Check for a `JMP` instruction
1	Hardware Breakpoint	Debugger detection by checking context flags

L. BENCHMARKING

Table VI presents the results of running a benchmark tool[42] with and without the monitor enabled. The `Base Value` column refers to the values obtained by running the system without the monitor. The `System Monitoring` column refers to the values obtained by running the monitor in a system-wide way, without disassembling instructions. The `Benchmark Monitoring` column refers to the data obtained by introspecting **and** disassembling benchmark instructions. All results were obtained by using the delayed data collection mode and running the monitor on a distinct core, the best usage scenario possible.

These results show us that unique operations are affected in distinct ways, due to the unique incidence of branch deviations. An example of such difference is observed between the floating-point tests and the integer ones. The MD5 calculation is also affected when the monitor is enabled, since it encompasses many branches due to algorithm's inner loops. We also notice the monitor imposes higher overheads when monitoring specific applications instead of the whole system. This result is expected since, besides the additional processing, some operations, such as memory read, may block. Additionally, in this scenario we observe a penalty in disk usage, due to log files being written.

[42]https://novabench.com/

Table VI: Benchmarking the system with and without the monitor.

Task	Base value	System monitoring	Penalty	Benchmark monitoring	Penalty
Floating-point operations (op/s)	101530464	99221196	2.27%	97295048	4.17%
Integer operations (op/s)	285649964	221666796	22.40%	219928736	23.01%
MD5 Hashes (hash/s)	777633	568486	26.90%	568435	26.90%
RAM transfer (MB/s)	7633	6628	13.17%	6224	18.46%
HDD transfer (MB/s)	90	80	11.11%	75	16.67%
Overall (benchm. pt)	518	470	9.27%	439	15.25%

Received Date 1; revised Date 2; accepted Date 3