PwIN – Pwning Intel piN: Why DBI is Unsuitable for Security Applications

Julian Kirsch(✉), Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel

Technical University of Munich, Munich, Germany
{kirschju,zhechev,bierbaumer,kittel}@sec.in.tum.de

Abstract. Binary instrumentation is a robust and powerful technique which facilitates binary code modification of computer programs even when no source code is available. This is achieved either statically by rewriting the binary instructions of the program and then executing the altered program or dynamically, by changing the code at run-time right before it is executed. The design of most Dynamic Binary Instrumentation (DBI) frameworks puts emphasis on ease-of-use, portability, and efficiency, offering the possibility to execute *inspecting* analysis code from an *interpositioned* perspective maintaining full access to the instrumented program. This has established DBI as a powerful tool utilized for analysis tasks such as profiling, performance evaluation, and prototyping.

The interest of employing DBI tools for binary hardening techniques (e.g. Program Shepherding) and malware analysis is constantly increasing among researchers. However, the usage of DBI for security related tasks is questionable, as in such scenarios it is important that analysis code runs *isolated* from the instrumented program in a *stealthy* way.

In this paper, we show (1) that a plethora of literature implicitly seems to assume *isolation* and *stealthiness* of DBI frameworks and strongly challenge these assumptions. We use Intel Pin running on `x86-64` Linux as an example to show that assuming a program is running in context of a DBI framework (2) the presence thereof can be detected, (3) policies introduced by binary hardening mechanisms can be subverted, and (4) otherwise hard-to-exploit bugs can be escalated to full code execution.

Keywords: Dynamic Binary Instrumentation · Intel Pin
Control Flow Integrity · Program shepherding · Malware analysis
Evasive malware · Virtual machine escape · Exploitation

1 Introduction

Malware continues to be a growing cyber security threat even nowadays. In the early days of the Internet malware was developed for mainly experimental reasons [26]. However, in recent years we are witnesses of malware utilized for theft of confidential data, denial-of-service of commercial systems, or even black mailing and cyber espionage. Industry and academia are constantly striving to develop countermeasures against these threats in form of advanced malware

J. Lopez et al. (Eds.): ESORICS 2018, LNCS 11098, pp. 363–382, 2018.
https://doi.org/10.1007/978-3-319-99073-6_18

detection approaches. However, malware developers continue to become more creative in their attempt to hinder the analysis of malware samples. Dynamic Binary Instrumentation (DBI) can help analysts to inspect applications' characteristics or alter their functionalities even when no source code is available. Therefore, DBI is easily employed as a malware analysis tool where the existence of anti-analysis techniques and the absence of source code are very common.

Similarly, computer systems are often subject to external attacks that aim to gain control over their functionality by leveraging malicious inputs. Such attacks attempt to trigger existing programming mistakes in software such as memory corruption bugs to subvert execution. DBI frameworks provide a possibility to conveniently add new functionalities to existing binaries, thus rendering these frameworks useful to harden software. One peculiarity, illustrating this approach, is *program shepherding* [17] – a technique that involves monitoring of all control transfers to ensure that each satisfies a given security policy, such as restricted code origins and controlling return targets. According to the program shepherding's paradigms this is possible because the hardened application is executed in the context of a DBI framework. A typical example of program shepherding is the implementation of Control Flow Integrity (CFI) policies using DBI to operate on Commercial Off-The-Shelf (COTS) binaries.

In this work we challenge both scenarios painted above. We argue that the original intent driving the motivation to build DBI frameworks was the ability to execute analysis code in a way that *interposes* execution of the instrumented program, *i.e.* analysis code can subscribe to be *notified* of any occurring event taking place in context of the instrumented program. Furthermore, an important design goal of DBI was to equip analysis code with full *inspection* capabilities covering the complete architectural state of the target. In practice this is typically achieved by introducing a single address space for both, analysis code and instrumented program.

This key observation is the main motivation behind our research. We show that due to the shared memory model, DBI frameworks in their current state are inherently incapable of providing neither *stealthiness* of the analysis code nor *isolation* of the analysis code against manipulations of the instrumented target. In our opinion, this *conceptionally renders them unsuitable for malware analysis and program shepherding.*

In a nutshell, this paper makes the following contributions:

Relevance. We identify DBI to be a common instrument for security-related tasks such as malware analysis and application hardening in literature.

Detectability. We demonstrate that it is trivial for an application to detect whether it is running in context of a DBI framework, enabling malicious software to behave in different ways during analysis.

Escapability. We attest that a malicious application can break out of the instrumentation engine and execute arbitrary code outside of the DBI framework.

Increased Attack Surface. We argue that counter-intuitively instead of *increasing* security by introducing DBI based software hardening measures, DBI actually *decreases* the overall security by escalating an otherwise hard-to-exploit real world bug (CVE-2017-13089) into full code execution.

2 Background and Related Work

In this chapter we discuss background about essential characteristics of DBI in general, introduce a consistent taxonomy used throughout this work, and discuss the usage of DBI frameworks for security in academic literature.

2.1 Dynamic Binary Instrumentation

A typical DBI framework consists of three components in a single address space:

1. The compiled target program which functionality should be altered
2. The functionality that is to be added to the target program
3. The DBI platform injecting the additional code into the target binary and ensuring proper execution.

Implementers typically develop their own *analysis plugins* which the *instrumentation platform* injects into the binary code of an application (*instrumented application*) that should be analyzed. The instrumentation platform exposes an API that enables the analysis plugin to register callbacks for certain events happening during the execution of the instrumented application. For example, it might be desirable for an analysis plugin implementing a *shadow stack* to receive a callback whenever the instrumented application tries to execute a `call` or `ret` instruction (*interposition*). Once the analysis plugin is notified (synchronously) of the execution of such an instruction, it may now freely inspect or modify all register and memory contents of the instrumented application (*inspection*).

2.2 Required Security Properties of Analysis Frameworks

In context of this work, we follow the taxonomy of Garfinkel and Rosenblum [14] to outline key requirements that any dynamic analysis framework needs to fulfill. In accordance to this work, we introduce analysis plugin and the instrumentation platform to form the *analyzing system*, as opposed to the instrumented application which constitutes the *analyzed system*. Then, the Garfinkel and Rosenblum taxonomy can be rephrased to DBI tools as follows:

R1 Interposition. *The analyzing system can subscribe to and is notified of certain events within the analyzed system.* For DBI this means that the instrumentation platform stops execution of the instrumented application and transfers control to the analysis plugin once certain events occur.

R2 Inspection. *The analyzing system has access to the full state of the analyzed system. Thus, the analyzed system is unable to evade analysis.* In context of our work this implies that the analysis plugin can freely access and modify all memory and register contents of the instrumented application.

R3 Isolation. *The analyzed system is unable to tamper with the analyzing system or any other analyzed system.* This means that the instrumentation platform and analysis plugin have to defend themselves against (malicious) modifications performed by the instrumented application.

In addition, researchers realized that dynamic analysis systems suitable to handle malware also need to operate in a way *transparent to the analyzed system.* This has the simple reason that so-called split personality malware might evade dynamic analysis if it is capable of detecting the analysis environment, as pointed out by Lengyel *et al.* [20]:

R4 Stealthiness. *The analyzed system is unable to detect if it currently undergoes analysis.* This means that the instrumented application must not be able to infer the presence of the instrumentation platform.

2.3 DBI Use in Literature

There are numerous examples of DBI utilization not only by the research community but also in commercial software development.

Binary Analysis. Many researchers develop DBI tools in order to perform analysis of binaries, *e.g.* Salwan *et al.* developed *Triton* [30], a concolic execution framework. Clause *et al.* [9] implement a dynamic taint analysis tool which supports data-flow and control-flow based tainting using DBI. Other analysis tools based on Intel Pin include a debugging backend shipped by default with the Interactive Disassembler (IDA) as well as *Lighthouse*[1], a coverage measurement tool created to enrich static analysis with dynamic information.

Bug Detection. Even in 2018, vulnerabilities resulting from memory corruption bugs [25] are still problematic. Many researchers implement vulnerability detection and prevention tools using DBI to limit the potential damage. This is the case because DBI provides them the advantage so that custom security code may be directly executed within the analyzed/hardened program. The Valgrind distribution includes a lot of profiling and debugging tools, such as *Memcheck* [22] which detects memory-management problems, as well as the heap profiler *Massif* [24]. Similarly, on the Windows family of Operating Systems (OSs) *Dr. Memory* [7] is a memory monitoring tool built on the DynamoRIO framework capable of identifying memory-related programming errors.

Program Shepherding/(CFI). A lot of research is recently conducted regarding program shepherding and CFI which attempts to restrict the set of possible control flow transfers to those that are strictly required for correct program execution [3]. In order to implement this approach, Davi *et al.* [10] developed a Pintool that dynamically enforces sanitizing return address checks by employing a shadow stack at run-time. While the idea of a shadow stack is much older [8,33], the advantage of this approach was the ease of development of the dynamic security enforcement tool. A similar approach was chosen by van der Veen *et al.* who developed a Linux kernel module and a Dyninst plugin [32] which both determine and restrict the valid execution paths and thereby ensure correct

[1] https://github.com/gaasedelen/lighthouse.

program execution. Instead of verifying the return address's validity, Tymburibá *et al.* [31] in contrast try to utilize Return-Oriented programming (ROP) gadgets' characteristics in order to prevent the hijacking of program's execution flow. In their Pintool called *RipRop* they detect unusually high rates of successive indirect branches during the execution of unusually short basic blocks, which may be an indication of a undergoing ROP attack. Later, in the same year Follner *et al.* present *ROPocop* [13], another Code-Reuse Attack (CRA) detection framework targeted at Windows `x86` binaries. It combines the idea of Tymburibá *et al.* together with a custom shadow stack and a technique which ensures no data is unintentionally executed. Yet another example of a Pintool utilized in ROP attack detection was proposed by Elsabagh *et al.* Their tool *EigenROP* attempts to detect anomalies in the execution process [11], due to execution of ROP gadgets, based on directional statistics and the program's own characteristics. Finally, Qiang *et al.* built a fully context-sensitive CFI tool [28] on top of Pin that may be used to protect COTS binaries. Among other advantages is that the tool checks the execution path instead of checking each edge in this execution path one by one which helps accelerate the process.

Malware Analysis. In addition, many security analysts employ DBI tools to study and profile malicious programs' behavior. Both to harden productive applications as well as to understand and reverse engineer potentially malicious program functionality in a sandbox environment. For instance, Gröbert *et al.* take advantage of a Pintool to generate execution traces and apply several heuristics to automate the identification of cryptographic primitives [15] in malicious samples. Kulakov developed a Pintool which performs static malware analysis in order to generate a loose timeline of the whole execution [19]. Additionally, he created an IDA plugin for better visualization of the data. Banescu *et al.* [4] proposed an empirical framework which is able to behaviorally obfuscate standard malware binaries. The program's observable behavior or path is defined by all internal computations and the sequence of accomplished system calls during its execution. In order to obfuscate malware samples, Banescu *et al.* [4] implemented a Pintool which inserts and reorders system calls into the binary without modifying its functionality but altering its known observable behavior.

Note that for the latter two of these domains, both Isolation and Stealthiness are a fundamental requirement to provide the proposed security guarantees.

2.4 Scope

To our perception, the most prominent examples of DBI frameworks nowadays are Intel Pin [21], Dyninst [5], Valgrind [23], DynamoRIO [6] and (more recently) QBDI [2] and Skorpio [29]. In the following, we focused (almost exclusively) on Intel Pin version 3.5 in Just-In-Time (JIT) mode on Linux while checking our results also against other common DBI implementations. We also utilize, as the time of writing, the latest release of Ubuntu 17.10 (64 bit) so that we can benefit

Table 1. Description of different DBI detection techniques. An asterisk (*) in the first column indicates a technique newly discovered during our research. All other techniques were adopted from their 32 bit versions targeting Windows presented in [12], except `enter` which is proposed by Ahmed Bougacha (See Footnote 3).

Technique	Type	Brief description
`envvar`	EA	Checks for Pin specific environment variables on stack
`enter`	CA	Checks whether `enter` instruction is legal and can be executed
`fsbase`*	CA	Checks if `fsbase` value is the same using `rdfsbase` and `prctl`
`jitbr`*	CO	Detects time overhead when a conditional branch is jitted
`jitlib`	CO	Detects JIT compiler overhead when a library is loaded
`nx`*	CA	Tries to execute code on a non-executable page
`pageperm`	EA	Checks for pages with `rwx` permissions
`mapname`	EA	Checks mapped files' names for known values (*pinbin*, *vgpreload*)
`ripfxsave`	CA	Executes `fxsave` instruction and checks the saved rip value
`ripsiginfo`*	CA	Causes an `int3` and checks the saved `rip` value in `fpregs`
`ripsyscall`	CA	Checks whether `rip` value is saved in `rcx` after a syscall
`smc`*	CA	Check whether the framework detects Self-Modifying Code
`vmleave`	EA	Checks for known code patterns (`VMLeave`)

from the latest security mechanisms, such as, for example, a higher number of randomized bits by Address Space Layout Randomization (ASLR)[2].

Note that from the previously defined requirements, R1 (Interposition) and R2 (Inspection) are fundamental features of DBI. In the following sections, we will challenge the previously defined requirements R3 (Isolation) and R4 (Stealthiness) and show that subversion of any thereof consequently also annihilates R1 (Interposition) and R2 (Inspection).

3 Stealthiness

In this section we present several techniques that reliably detect the presence of different DBI frameworks. To achieve this, we not only adopted several existing DBI detection techniques [12] to Linux `x86-64` but also found new, previously

[2] See `/proc/sys/vm/mmap_rnd_bits`.

Table 2. Detection mechanisms on different DBI frameworks. A ϟ indicates that the test reliably detects the presence of the indicated DBI framework, a ✗ means that a particular test does not detect the presence of the respective DBI framework.

Technique	Type	Pin	Valgrind	DynamoRIO	QBDI
`jitbr`	CO	ϟ	ϟ	ϟ	✗
`jitlib`	CO	ϟ	ϟ	ϟ	✗
`pageperm`	EA	ϟ	ϟ	ϟ	ϟ
`vmleave`	EA	ϟ	✗	✗	ϟ
`mapname`	EA	ϟ	ϟ	ϟ	✗
`smc`	CA	ϟ	ϟ	✗	ϟ
`ripfxsave`	CA	ϟ	ϟ	✗	ϟ
`ripsiginfo`	CA	ϟ	ϟ	ϟ	ϟ
`ripsyscall`	CA	ϟ	✗	ϟ	ϟ
`nx`	CA	ϟ	ϟ	ϟ	ϟ
`envvar`	EA	ϟ	ϟ	ϟ	✗
`fsbase`	CA	ϟ	–	ϟ	✗
`enter`	CA	✗	ϟ	✗	✗

unknown detection techniques. We group detection techniques in three categories; (1) code cache/instrumentation artifacts (CA), (2) JIT compiler overhead (CO), and (3) runtime environment artifacts (EA). In this paper we only describe techniques from categories (1) and (3) in detail. While we explain these techniques on Pin, we found them also applicable to other DBI implementations.

We have developed a tool called *jitmenot* which employs 13 different DBI detection mechanisms summarized in Table 1, 7 of which were adopted from their Windows specific 32 bit counterparts presented elsewhere [12] and one was proposed by Ahmed Bougacha[3]. In the following, we describe only the most prominent examples for space reasons. Our testing tool *jitmenot* is released under an open-source license and can be downloaded from GitHub[4]. See Table 2 for an overview of which detection technique is able to detect which of the analyzed DBI frameworks.

3.1 Code Cache/Instrumentation Artifacts

In the first category – code cache artifacts – we include anomalies introduced by the fact that the executed code is not the original one.

Abusing the `syscall` Instruction (`ripsyscall`). One less known property of the x86-64 architecture is that when executing any system call via the `syscall` instruction, the current instruction pointer value is copied to the `rcx` register [16], such that the kernel can restore execution correctly via the `sysret` instruction later. As operation of the OS's kernel happens transparently, user land perceives

[3] http://repzret.org/p/detecting-valgrind.
[4] https://github.com/zhechkoz/PwIN.

the `syscall` instruction to have the side effect of setting the `rcx` register to the instruction right behind the `syscall`. The `ripsyscall` method involves the way the DBI frameworks emulate system calls. For example, when Pin has to accomplish some task outside of the Virtual Machine (VM), such as forwarding a system call request from the instrumented application or determining the next basic block to execute, the register state of the instrumented application is saved and the VM is left.

However, this is not the case for an instrumented application executed within DBI. Since, DBI frameworks wrap all system calls performed by the instrumented application, they need to save the program's register state before switching from the context of the instrumented application to its own internal state. When re-entering the context of the instrumented application, apart from the system call's result in `rax`, no other side effects are propagated back to the program. As a result, the `rcx` register observed by the instrumented application stays constant across system calls. This discrepancy can be used as a detection mechanism.

Self-modifying Code (`smc`). Yet another code cache artifact involves the way DBI frameworks handle Self-Modifying Code (SMC) together with the fact that instrumentation is done at basic block granularity. According to Intel, the Pin framework, for example, does attempt to detect manipulations of the original code of the instrumented application by exposing the `PIN_Set-Smc-Support` configuration option and a corresponding callback function `TRACE_-AddSmc-Detected-Function`. However, the analysis plugin programmer has to manually trigger code cache invalidation upon receiving a SMC notification to re-trigger the JIT compiler for the altered code. If the analysis plugin programmer does not handle SMC, or does not invalidate the code cache, the instrumented application could detect the presence of Pin as follows: First, the instrumented application marks its own code as readable, writeable and executable prior to executing a probe instruction once, making sure it gets placed into the code cache. Then the malicious tool modifies the immediate operand of the probe `mov` instruction from I_0 to I_1 in the code cache. Since Pin does not automatically invalidate the code cache only the original code is modified, resulting in `mov` ending up with immediate operand I_0. If the same sequence is executed outside of a instrumentation platform, the code change takes effect immediately and the `mov` instruction will use I_1 as immediate operand. Only if the analysis plugin monitors all write accesses of the application to its own text segment it can reliably detect SMC. Furthermore, a code cache invalidation request after every write (incurring performance overhead) is needed to prevent the attack sketched above.

Wrong Emulation of `enter` Instruction (`enter`). Some DBI frameworks, such as Valgrind, first translate the program into a processor-neutral Intermediate Representation (IR), which is then instrumented by the analysis plugin and in the end compiled to machine code. This implies that the DBI framework is capable of emulating the whole instruction set of the processor. However, since some instructions are less frequently used than others, DBI developers choose to either partially or completely not support them. An example of such an instruction is

the `x86 enter` instruction [16], which creates a stack frame for a procedure. This instruction executes as expected in a non-instrumented environment. However, when a program instrumented by Valgrind attempts to execute `enter`, a signal is raised because this particular instruction is not implemented in the IR. By catching this signal, an application can determine whether it is instrumented or not. Note that this behavior is not observed in Intel Pin since it does not rely on IR for instrumentation.

Neglecting No-eXecute Bit (`nx`). W⊕X is an exploitation mitigation technique enabling the OS to mark writeable pages in memory as not executable. The consistent application of W⊕X denies an attacker the ability to introduce own code into the address space of a program before transferring the execution flow to it. However, when the JIT compiler of a DBI framework fetches new instructions for instrumentation, it does not check whether the source memory is marked as executable; as long as the page is readable the JIT compiler will translate any data present and emit executable assembly instructions. Note that **all** DBI frameworks we tested were vulnerable to this problem. Clearly, this is a huge security issue, as this implies that **any program** instrumented by a **DBI** framework **effectively** has **W⊕X disabled**. We utilized this fact as a detection technique in the following way: (1) Allocate a new page on the heap without execute permissions and place valid code in it, (2) then execute it. Without instrumentation, on any modern OS, a program trying to execute code on a page without `x` permissions will result in a crash. Otherwise, if the program is being instrumented, the program will be allowed to continue. This difference allows us to determine whether an application is currently instrumented or not.

Therefore, if it is possible to divert execution to a user-controllable buffer, an attacker can place shellcode in it and the VM will execute it. This effectively violates the Isolation property of DBI. Later we present a real-world example of how this can be leveraged to achieve a fully working exploit. This classifies as a major vulnerability issue not only in PinDBI framework but also in all other DBI engines which we tested as can be seen in Table 2. Nevertheless, introducing proper checks for correct memory page permissions before fetching code from memory could resolve this issue. This can be accomplished either in the JIT compiler or as a temporary fix integrated in the instrumentation platforms.

Real Instruction Pointer (`ripfxsave`/`ripsiginfo`). This technique was already introduced by Falcón and Riva [12]. However, as detection of the real `rip` also is a building block for attacks (against the DBI Isolation property) described later in this paper, we briefly summarize the techniques nevertheless.

In a nutshell, the DBI framework VMs execute only the translated and instrumented code of the application residing in the code cache but never the original code in the original text segment of the instrumented application. To maintain compatibility with non-relocatable applications, Pin attempts to mask the VM's `rip` with the instrumented program's original `rip` value whenever necessary.

One technique for finding the real `rip` abuses the systems Floating Point Unit (FPU): First, any FPU instruction (*e.g.* `fldz`) is executed. Afterwards, the FPU state is saved using the `fxsave` instruction. This state includes the

address of the most recently executed FPU instruction, which is not masked by any instrumentation platform. A malicious instrumented application can then compare this value to the (masked) value when directly reading the `rip` register. If the values do not match, the application is being instrumented.

The same can be achieved on Linux with the help of the OS: The `sigaction` system call allows the calling process to examine and/or specify the action to be associated with a specific signal. When configured, the OS provides information about the context at the time the signal occurred. If we execute any floating-point instruction and then initiate a signal (for instance by deliberately triggering a segmentation fault), the context argument passed to the signal handler contains the state of the FPU registers. By comparing these values with the corresponding `rip`, one can again determine whether the program is being instrumented.

3.2 Environment Artifacts

In this section we identify environmental artifacts introduced by DBI frameworks. By this we refer to anomalies in the execution environment. For example, the memory layout varies drastically with the usage of instrumentation platforms. Searching for DBI artifacts such as strings or particular code patterns in memory reveals the presence of a DBI tool inside the target process memory.

Pin Internal Environment Variables. Firstly, Pin expects certain environment variables in order to run properly (e.g. `PIN_INJECTOR64_LD_LIBRARY_PATH`). Because the framework spawns the instrumented process as its own child, these inherited environment variables can still be found in the instrumented application's memory. Querying them by means of `getenv` can therefore expose the underlying JIT engine.

Page Permissions and Code Patterns. A side effect of the JIT engine is the presence of pages that are marked simultaneously as readable, writeable, and executable. An attacker can use this knowledge to scan the address space for such pages (e.g. using `/proc/self/maps` on Linux). Knowing that `rwx` memory is not present in a particular binary this discrepancy reveals the presence of the JIT compiler' code cache.

Additionally, an attacker can scan the whole memory for mapped pages and compare names of memory mapped files with names related to the DBI framework (*e.g. pinbin* for Pin). Moreover, as the instrumented application and the analysis plugin share one address space, the full implementation of the instrumentation platform has to be present as well. For example, an attacker can search for specific code patterns such as the characteristic implementation of the transition between the instrumented application and the analysis plugin (`VMLeave`). This code sequence is distinctive because it saves the current VM context on the stack and restores the jitted application's registers, generating assembly code that can easily be recognized.

Another method revealing the JIT engine's presence relies on an egg-hunting approach: A simple loop might scan all pages for a unique 8-byte-pattern. Under normal circumstances, the pattern should be observed exactly once in the address space (the operand of the comparison). However, when executed in context of a DBI framework, the pattern emerges at least twice: Once in the original code and once in the code cache.

3.3 Summary

As one can see, an instrumented application can notice whether it is currently being executed in a DBI environment. By nature, JIT compilers cause a lot of noise which is not only hard to disguise but trying to do so introduces even more irregularities in the instrumented program execution (cf. [27]). It follows that, the requirement R4 (Stealthiness) which is essential for security applications such as malware analysis cannot be hold by DBI frameworks.

4 Isolation

After discussing detectability of DBI frameworks, the following section focuses on the methods and possibilities to escape from and consequently evade the instrumentation. In the original work describing Pin [21] in Sect. 3.3.1 the authors state that the instrumented application's code is never executed – instead it is translated (from machine instructions to the same kind of machine instructions) and executed together with the analysis plugin's procedures within a custom virtual environment (the Pin VM). All executed machine instructions reside in the VM (code cache) and the effect of any instruction cannot *escape* from the VM region. Like other VMs, the Pin framework manages the instrumented program's instruction pointer and translates each basic block of the original code lazily (*i.e.* when reached by the execution flow). Two properties make Pin subject to attacks compromising isolation: First, the VM may and will reuse already compiled code because of optimization benefits. Second, Pin does not employ any integrity checks of already translated instructions in the code cache. Therefore, we can alter already executed instructions in memory, as they (comfortably) reside on pages marked `rwx` by the instrumentation platform. Experimental evidence from Sect. 3 indicates that the code cache implemented by other DBI tools behaves in accordance with Pin's code cache. However, we target the DBI implementation of Pin on `x86-64` Linux in the following sections.

For this we distinguish two different attacker models, and describe an escaping mechanism suitable for each.

A1 Control of Code and Data. This is the most potent attacker. She can freely specify which code is executed in the instrumented application and is able to freely interact with the application while instrumented. In reality, such an attacker would craft a malicious binary in the hope that an analyst would execute the binary in a instrumentation platform.

A2 Control of only Data. This is the weaker of the two attacker models. In this case, an attacker only possesses copies of the instrumented application, instrumentation platform, analysis plugin, and all depending dynamic libraries. However, this attacker is also able to freely interact with the application containing memory corruption vulnerabilities while executed in an DBI framework. In practice this is the case when some binary hardening policy implemented using DBI gets attacked over the network.

While detectability always required an attacker of type **A1**, we show that it is possible for an attacker of type **A2** to escape from the instrumentation if the attacked program contains what is commonly referred to as a write-where-what vulnerability.

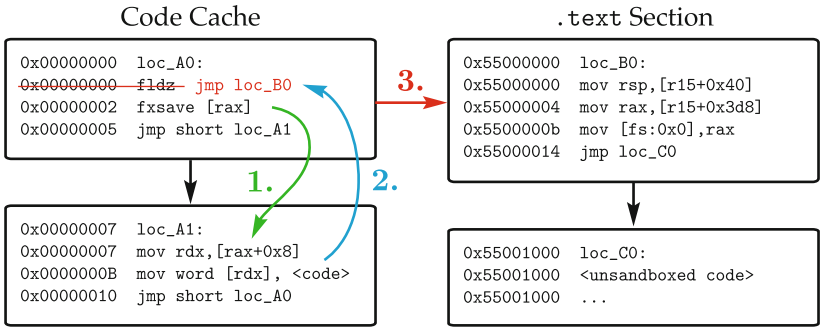

Fig. 1. A minimal program escaping from the Pin VM.

4.1 Escaping from Pin's Instrumentation Using Direct Code Cache Modification

First, we describe the escaping technique for the more potent attacker **A1** whose goal is to execute arbitrary code without Pin's instrumentation engine being able to embed callbacks notifying the analysis plugin. The existence of the just-in-time compilation allows us to first execute a basic block in order to allow the Pin VM to translate its assembly code and place its address in an internal hash table to find it later. Then the instrumented program can find the translated version of the basic block in the code cache (using the real instruction pointer detection techniques described in the previous section). It can then modify the jitted code arbitrarily. Once the execution flow reaches the modified basic block a second time, Pin will effectively execute whatever an attacker placed there. Figure 1 depicts the steps needed.

Prior to escaping from the VM, one first has to use any of the techniques to find the real `rip` value discussed in Sect. 3 (Block `loc_A0` in Fig. 1 showing the `ripfxsave` technique). As expected, Pin executes these instructions within its

own code cache. As a result, at the end of block `loc_A0`, `rax` now points to the FPU context containing a pointer to the beginning of `loc_A0`. Then (step **1.**), execution is redirected to block `loc_A1` using a `jmp` instruction, where an attacker places code that *patches out* the first instruction of `loc_A0` and replaces it with a control flow change eventually reaching `loc_B0` (step **2.**). Then, when the control flow reaches `loc_A0` for the second time, the modified instructions placed there will be executed, now redirecting execution to block `loc_B0` *residing in the original code* (step **3.**). As the code cache is mapped `rwx`, this does not trigger any page fault, hence the instrumentation engine does not get notified of the breach happening in the VM. To maintain ABI compatibility to arbitrary code embedded into the malicious executable, block `loc_B0` needs to restore the `rsp` and `fsbase` registers, which, due to the code generation strategy of the JIT compiler are conveniently accessible via a structure pointed to by register `r15`. Now, execution can move on to any arbitrary code `loc_C0` in the original executable prepared by the attacker – as all pages are mapped executable there is no mechanism allowing Pin to re-trigger the JIT compiler process to embed its instrumentation hooks. In fact, from Pin's perspective the application is still executed in the VM and awaits to regain control again, which never happens.

4.2 Escaping from Pin's Instrumentation Using an Existing Memory Corruption

As previously stated, it is also possible under certain circumstances to evade the instrumentation if only an attacker of type **A2** is present. Escaping the Pin sandbox in Linux without necessarily knowing any code cache address is also possible: We measured the relative offsets between all mapped pages in different executions of an application instrumented by Pin. As it can be seen in Fig. 3 (top right on page 18), the offset between libc and the code cache, as well as pinbin (main Pin binary) and Pin's own stack is constant. Leaking addresses from any of these code regions therefore allows us to reliably find the other mappings. Consequently, we can utilize all gadgets present in the code basis to build ROP chains, or directly write shellcode using a write-what-where vulnerability into the code cache. This is due to the fact that, as already explained, the Pin framework copies itself into the application's memory by allocating memory using `mmap`. As pointed out in earlier work [18], the addresses of consecutively allocated memory allocations returned by `mmap` are predictable (*i.e.* relative distances remain constant) in Linux. Thus, all required information can be calculated a priori based on known binaries of Pin, the analysis plugin, the instrumented application, and all dynamic link libraries (cf. Fig. 3 in the Appendix).

Since Pin does not monitor its code cache for external changes and does not restrict its execution to known memory locations, one can alter the instrumented processes memory in any suitable way. Moreover, the address of the code cache in the Linux version of Pin can be calculated by using any leaked address from other similarly created memory region. Therefore, if the binary contains a function that is executed twice and after its first invocation, a malicious user overwrites this function's instructions in the code cache, they are able to gain full control

over the application. Unfortunately, such a function (`rtld_lock_default_lock`) is contained within the dynamic loader, a core component of the Linux OS.

5 Increased Attack Surface

Previously we have shown that DBI frameworks are both detectable and escapable rendering them as not suitable for binary hardening or malware analysis. In this section, we show how implementing security mechanisms enforced by executing a given COTS binary in a DBI environment even introduces more possibilities to exploit already present bugs (i.e. attack surface is *increased* instead of *decreased*). To support this claim we discuss an example where a vulnerability that is not trivial to exploit during normal execution *becomes exploitable* when executed within a DBI framework interacting with an attacker of type **A2**.

5.1 The Return of Aleph One

During the study of detectability properties of instrumentation platforms we already pointed out that they fail to check the permissions of the code that is to be processed by their JIT engines. This means *any* data in memory can (and will) be translated to executable instructions if reached by the control flow. This transfers us back to the dawn of buffer overflows and shellcode execution era. As a simple example we can run an application which jumps to shellcode on the stack. Normally, because of the set No-eXecute bit in the page tables of the stack, the program would crash as soon as the instruction pointer points to an address on the stack. However, instrumenting the same binary with Pin does not crash the application. In fact, the execution continues and opens a shell.

5.2 Turning CVE-2017-13089 to a Code Execution Bug with the Help of Intel Pin

To underline the exploitability claim, we have implemented a Proof Of Concept binary (*PwIN*) that exploits an existing CVE vulnerability (CVE-2017-13089, cf. [1]) that is not easily exploited when executed in a normal environment. CVE-2017-13089 is a bug in *wget* versions older than 1.19.2 found in `http.c:skip_short_body()`. The bug itself is described in more detail in the next section. Without Intel Pin the strongest attack (known to us) results in a $\frac{1}{16}$ probability of leaking an arbitrary file stored on the victim to the server (see below). We will discuss how the same bug can be escalated to full code execution if the victim is instrumented using Intel Pin.

Description of the Bug. The vulnerable function in *wget* is called when processing HTTP redirects together with HTTP chunked encoding. The chunk parser uses `strtol()` to parse each chunk's length into a variable of type `long`. Prior to copying a chunk's contents into a buffer on the stack, the code validates

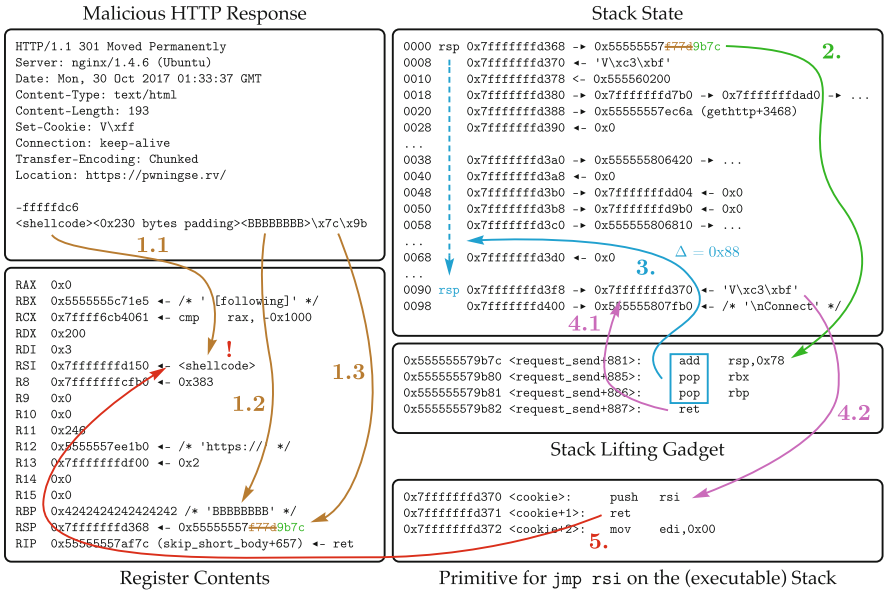

Fig. 2. Control flow and state changes of *wget* when attacked by a malicious server. The last control transfers (**4.2** in purple and **5.** in red) mark the transitions that are enabled by the usage of Pin. Under normal circumstances, the program would crash as the buffers on the stack containing the malicious shellcode would not be executable. (Color figure online)

that the chunk size specified in the HTTP request fits into the buffer, forgetting to ensure the supplied signed value is actually a *positive* number. The code then tries to skip the chunk in pieces of 512 bytes but ends passing a negative length to `connect.c:fd_read()`. Unfortunately, `fd_read()`'s length argument is of type `int`, thus the high 32 bits of the length variable are discarded. Therefore, values in the range `0xffffffff00000000` to `0xffffffffffffffff` pass all checks while the truncation to a 32 bit value still allows an attacker to control the length of the read chunk and to overflow the `dlbuf` variable, a buffer of fixed size, on the stack.

Exploitation of the Bug. The bug allows for a continuous write of arbitrary data on the stack. Due to the absence of stack canaries, the saved return address on the stack can be compromised. However, without the knowledge of the current state of ASLR, there is not much an attacker can do, as she does not know any pointer pointing into valid memory (the binary is compiled as position independent executable). Consequently, the only remaining option to continue exploitation is a *partial pointer override*. With this technique, an attacker abuses the fact that ASLR operates at a page ($4096 = 2^{12}$ bytes) granularity. Therefore, the lowest 12 bits of any object within the address space are deterministic.

As a consequence, an attacker can now *trade* the number of ROP gadgets reachable by a `ret` for exploit reliability by overwriting parts of the saved return pointer on the stack. For example, a two-byte partial pointer overwrite needs to guess $2 \cdot 8 - 12 = 4$ bits of randomness, allowing to transfer control to a region sharing the same $2^{2 \cdot 8} = 65536$ bytes region with the original return address. Automatically evaluating all targets within this region using dynamic analysis does not unveil any target where an attacker could trivially obtain arbitrary code execution. The only noteworthy effect that can be observed is when targeting `body_file_send()`, as register allocation (cf. Fig. 2) matches the signature of this function with `rsi` pointing to attacker controlled data specifying a file name to transfer from the client to the server.

However, when running in context of Intel Pin we can inject and execute shellcode situated in non-executable memory regions, reducing the challenge of achieving code execution to *just* having to find a reliable mechanism to jump to a pointer to data we control. Our full exploit chain is visualized in Fig. 2: Fortunately, when reaching the end of the `skip_short_body()` function the `rsi` register (step **1.1**) contains the address of `dlbuf` (controlled by the attacker). However, there are no convenient gadgets reachable with a partial overwrite on the return address which may divert the code execution to the address contained in `rsi`. We remedy this by injecting our own `jmp rsi` gadget into a buffer that we can divert control to using the partial overwrite in step **1.3**. As expected, before reaching the return pointer on the stack, we inevitably have to load an invalid pointer to `rbp` register (step **1.2**) which fortunately, does not negatively influence our future actions. We can reach a stack lifting gadget with a partial overwrite (step **2.**) that increments the stack pointer by $\Delta = 0x88$ bytes (step **3.**). The new stack pointer location now points to a pointer to the *UTF-8* encoded value of the contents of the `Set-Cookie` header of the HTTP response. At this point the `ret` will transfer control to an attacker controlled buffer (steps **4.1** and **4.2**) but the UTF-8 encoding constrains the shellcode in an uncomfortable way. Luckily enough, the string `V\xff` is encoded to `V\xc3\xbf` which is perfectly valid UTF-8 *and* disassembles to `push rsi; ret` at the same time. As `rsi` still points to (now unconstrained) attacker controlled shellcode from the HTTP response body residing in `dlbuf`, this control transfer (step **5.**) is the last step in achieving code execution. This attack succeeds with a probability of $\frac{1}{16}$, due to the partial pointer override used in the first step.

6 Discussion

We conclude this work by inviting other researchers to reproduce our findings, discussing limitations and future improvements.

6.1 Reproducibility and Source Code Availability

In the spirit of open research and to enable others to reproduce our findings, we publish the source code of all the experiments conducted in context of this work. A Docker container is available via download at https://kirschju.re/pwin.

6.2 Limitations and Future Work

There are some known limitations associated with the proposed methodologies. First, the detect mechanism can be avoided by extending DBI frameworks. However, we already discussed that the JIT compiler introduces a lot of noise in the normal execution of the application, constantly turning up new possibilities to detect irregularities. Secondly, to escape the DBI sandbox, the application has to execute a function at least twice and simultaneously know its address in the code cache. Moreover, it needs to alter this function in the code cache before its last execution. Additionally, to escape DBI without knowing exactly the address of a function executed at least twice, requires calculation of the necessary memory locations. This is only possible, since the offset between pages created by `mmap` is constant [18]. However, the attacker has to possess copies of the instrumented application, instrumentation platform, analysis plugin, and all depending dynamic libraries because they all reside in the code cache and influence the fixed offsets between memory regions.

The research presented in this work discussed in detail the disadvantages of utilizing DBI engines in the security domain. The most fundamental problem is that DBI logic and application reside in the same address space, with no isolation present. The question remains how Intel Pin and other DBI frameworks can mitigate this problem in the future and how these techniques would influence our research. A possible mitigation strategy might introduce Intel Memory Protection Keys to change memory access permissions from user space without sacrificing performance.

6.3 Conclusion

In this paper, we showed that DBI frameworks are commonly used in a context of security, both as an analysis platform, as well as a hardening tool. Thus we systematically discussed the requirements for DBI frameworks to be used within such a context. We showed, that DBI is not able to hold these requirements in practice. We demonstrate, that the stealthiness requirement does not hold in practice by enumerating different inherent techniques to detect DBI. In addition, we also attested that DBI does not sufficiently isolate instrumented applications from the instrumentation framework, which provides a possibility for instrumented applications to gain arbitrary code execution on the analysis system. Finally, we argue, that instead of *increasing* security by introducing DBI based software hardening measures, DBI actually *decreases* the overall security by escalating an otherwise hard-to-exploit real world bugs into to full code execution. To support our claim, we implemented a couple of Proof Of Concepts to support our claims, which we are happy to freely share with the community.

A Appendix

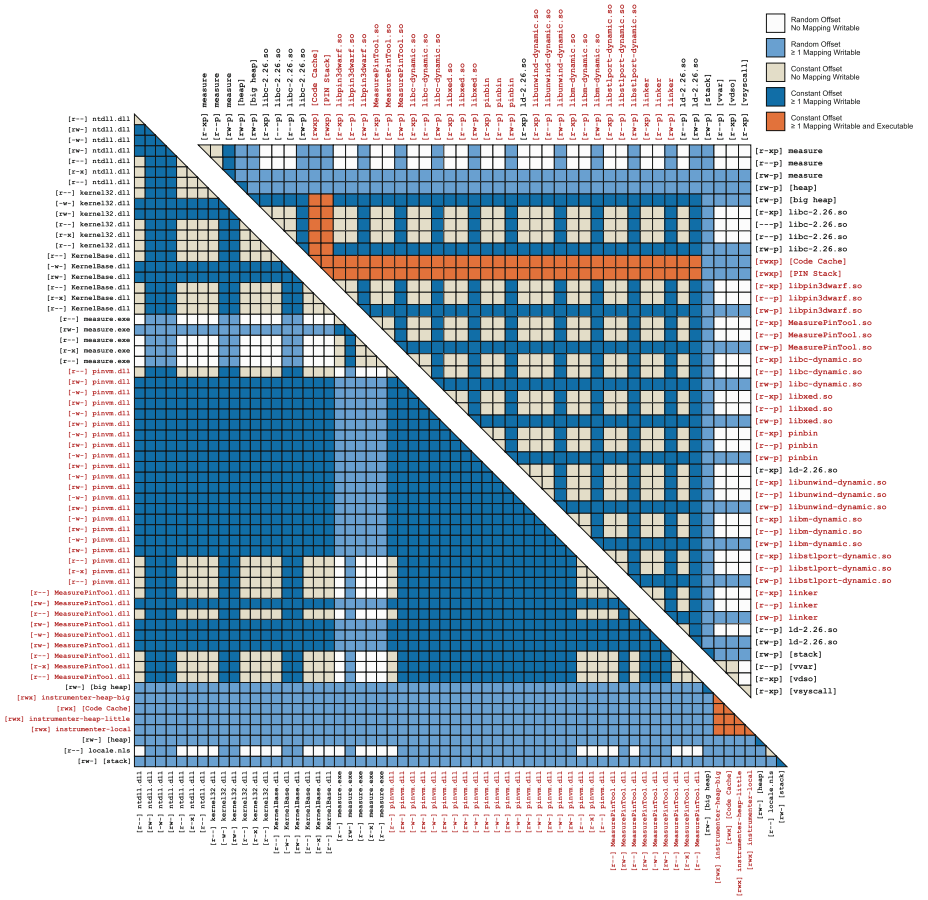

Fig. 3. Color matrices showing **memory regions** sharing random (□/■) or constant (□/■/■) **distances** with each other for applications instrumented by Linux (above right) and Windows (down left) version of Pin. The region names in **red** are additional components added by the instrumentation framework while in **black** are presented the program's original mapped files. (Color figure online)

References

1. CVE-2014-0160. Available from MITRE, CVE-2017-13089. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13089. Accessed 24 Apr 2018
2. QuarkslaB Dynamic binary Instrumentation (QBDI). https://qbdi.quarkslab.com/. Accessed 24 Apr 2018
3. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. **13**, 4:1–4:40 (2009)
4. Banescu, S., Wüchner, T., Guggenmos, M., Ochoa, M., Pretschner, A.: FEEBO: an empirical evaluation framework for malware behavior obfuscation. arXiv preprint arXiv:1502.03245 (2015)
5. Bruening, D., Duesterwald, E., Amarasinghe, S.: Design and implementation of a dynamic optimization framework for windows. In: 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4) (2001)
6. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In: International Symposium on Code Generation and Optimization, CGO 2003, pp. 265–275. IEEE (2003)
7. Bruening, D., Zhao, Q.: Practical memory checking with Dr. Memory. In: Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, pp. 213–223. IEEE Computer Society (2011)
8. Chiueh, T.c., Hsu, F.H.: RAD: a compile-time solution to buffer overflow attacks. In: 21st International Conference on Distributed Computing Systems, pp. 409–417. IEEE (2001)
9. Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis, pp. 196–206. ACM (2007)
10. Davi, L., Sadeghi, A.R., Winandy, M.: ROPdefender: a detection tool to defend against return-oriented programming attacks. In: ASIACCS (2011)
11. Elsabagh, M., Barbará, D., Fleck, D., Stavrou, A.: Detecting ROP with statistical learning of program characteristics. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 219–226. ACM (2017)
12. Falcón, F., Riva, N.: Dynamic binary instrumentation frameworks: i know you're there spying on me. In: RECon 2012 (2012). https://recon.cx/2012/schedule/attachments/42_FalconRiva_2012.pdf. Accessed 25 Apr 2018
13. Follner, A., Bodden, E.: ROPocop - dynamic mitigation of code-reuse attacks. J. Inf. Secur. Appl. **29**, 16–26 (2016)
14. Garfinkel, T., Rosenblum, M., et al.: A virtual machine introspection based architecture for intrusion detection. In: NDSS, vol. 3, pp. 191–206 (2003)
15. Gröbert, F., Willems, C., Holz, T.: Automated identification of cryptographic primitives in binary programs. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 41–60. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_3
16. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, January 2018
17. Kiriansky, V., Bruening, D., Amarasinghe, S.P.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206. USENIX Association, Berkeley (2002)
18. Kirsch, J., Bierbaumer, B., Kittel, T., Eckert, C.: Dynamic loader oriented programming on Linux. In: ROOTS (2017)

19. Kulakov, Y.: MazeWalker - enriching static malware analysis. In: RECon 2017 (2017). https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-MazeWalker.pdf. Accessed 25 Apr 2018
20. Lengyel, T.K., Maresca, S., Payne, B.D., Webster, G.D., Vogl, S., Kiayias, A.: Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In: Proceedings of the 30th Annual Computer Security Applications Conference, pp. 386–395. ACM (2014)
21. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM Sigplan Notices, vol. 40, pp. 190–200. ACM (2005)
22. Nethercote, N., Seward, J.: How to shadow every byte of memory used by a program. In: VEE (2007)
23. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan Notices, vol. 42, pp. 89–100. ACM (2007)
24. Nethercote, N., Walsh, R., Fitzhardinge, J.: Building workload characterization tools with Valgrind. In: IISWC (2006)
25. One, A.: Smashing the stack for fun and profit. In: Phrack 49 (1996)
26. Orman, H.: The Morris worm: a fifteen-year perspective. IEEE Secur. Priv. **99**(5), 35–43 (2003)
27. Polino, M., et al.: Measuring and defeating anti-instrumentation-equipped malware. In: Polychronakis, M., Meier, M. (eds.) DIMVA 2017. LNCS, vol. 10327, pp. 73–96. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-60876-1_4
28. Qiang, W., Huang, Y., Zou, D., Jin, H., Wang, S., Sun, G.: Fully context-sensitive CFI for COTS binaries. In: Pieprzyk, J., Suriadi, S. (eds.) ACISP 2017. LNCS, vol. 10343, pp. 435–442. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59870-3_28
29. Quynh, N.A.: Skorpio: advanced binary instrumentation framework. In: OPCDE 2018, Dubai, April 2018
30. Saudel, F., Salwan, J.: Triton: a dynamic symbolic execution framework. In: Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, 3–5 June 2015, pp. 31–54. SSTIC (2015)
31. Tymburibá, M., Emilio, R., Pereira, F.: RipRop: a dynamic detector of ROP attacks. In: Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice, p. 2 (2015)
32. van der Veen, V., et al.: Practical context-sensitive CFI. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 927–940. ACM (2015)
33. Vendicator, S.S.: A Stack Smashing Technique Protection Tool for Linux (2000). http://www.angelfire.com/sk/stackshield/info.html. Accessed 24 Apr 2018