Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks

AILTON SANTOS FILHO, Institute of Computing, Federal University of Amazonas (UFAM)
RICARDO J. RODRÍGUEZ, Department of Computer Science and Systems Engineering, University of Zaragoza
EDUARDO L. FEITOSA, Institute of Computing, Federal University of Amazonas (UFAM)

Dynamic Binary Instrumentation (DBI) is a dynamic analysis technique that allows arbitrary code to be executed when a program is running. DBI frameworks have started to be used to analyze malicious applications. As a result, different approaches have merged to detect and avoid them. Commonly referred to as *split personality malware* or *evasive malware* are pieces of malicious software that incorporate snippets of code to detect when they are under DBI framework analysis and thus mimic benign behavior. Recent studies have questioned the use of DBI in malware analysis, arguing that it increases the attack surface. In this article, we examine the anti-instrumentation techniques that abuse desktop-based DBI frameworks and existing countermeasures to determine if it is possible to reduce the exploitable attack surface introduced by these DBI frameworks. In particular, we review the related literature to identify (i) the existing set of DBI framework evasion techniques and (ii) the existing set of countermeasures to avoid them. We also analyze and compare the taxonomies introduced in the literature, and propose a new taxonomy that expands and completes the previous taxonomies. Our findings demonstrate that despite advances in DBI framework protections that make them quite suitable for system security purposes, more efforts are needed to reduce the attack surface that they add during application analysis. Only 12 of the 26 evasion techniques covered in this document have countermeasures, threatening the transparency of DBI frameworks. Furthermore, the impact in terms of performance overhead and effectiveness of these countermeasures in real-world situations is unknown. Finally, there are only proofs of concept for 9 of these 26 techniques, which makes it difficult to validate and study how they evade the analysis in order to counter them. We also point out some relevant issues in this context and outline ways of future research directions in the use of DBI frameworks for system security purposes.

CCS Concepts: • **Security and privacy** → Systems security; **Software and application security**; **Malware and its mitigation;**

Additional Key Words and Phrases: Dynamic binary instrumentation, analysis evasion, analysis-aware malware

The research of A. Santos Filho and E. L. Feitosa, according to Article 48 of Decree nº 6.008/2006, was partially funded by Samsung Electronics of Amazonia Ltda, under the terms of Federal Law nº 8.387/1991, through agreement nº 003/2019, signed with ICOMP/UFAM, and was supported by the Coordination for the Improvement of Higher Education Personnel - Brazil (CAPES) - Finance Code 001. The research of R. J. Rodríguez was supported in part by the Spanish Ministry of Science, Innovation and Universities under grant MEDRESE-RTI2018-098543-B-I00, by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R), and by the University of Zaragoza and the *Fundación Ibercaja* under grant JIUZ-2020-TIC-08.

Authors' addresses: A. S. Filho and E. L. Feitosa, Institute of Computing, Federal University of Amazonas (UFAM), Av. General Rodrigo Octavio Jordão Ramos, 1200, Coroado I, Manaus - AM, Brazil, 69067-005; emails: {assf, efeitosa}@icomp.ufam.edu.br; R. J. Rodríguez, Dept. of Computer Science and Systems Engineering, University of Zaragoza, Ed. Ada Byron, Calle María de Luna 1, Zaragoza, Spain, 50018; email: rjrodriguez@unizar.es.

2576-5337/2022/02-ART11 $15.00
https://doi.org/10.1145/3480463

ACM Reference format:
Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. 2022. Evasion and Countermeasures Techniques to Detect Dynamic Binary Instrumentation Frameworks. *Digit. Threat.: Res. Pract.* 3, 2, Article 11 (February 2022), 28 pages.
https://doi.org/10.1145/3480463

1 INTRODUCTION

Nowadays, software specially crafted with malicious intentions (commonly referred to as *malware*) remains a growing and relevant problem for the security of modern computer systems. This fact supposes a tremendous issue for the anti-malware companies that need to provide an up-to-date database of known threats to protect their customers in a timely manner, since malware analysts face every day with an increasing number of malware samples. For instance, Kaspersky analyzed more than 360,000 malware samples per day in 2017 [23]. Likewise, statistics from McAfee Labs [36] indicate that 50M malicious samples were analyzed in the last quarter of 2018, accumulating a total of more than 800M malware samples analyzed in all of 2018. A recent article [71] also provides a similar number of malware samples analyzed annually, counting over 294M of samples targeting Windows environments only.

Given the evolution in quantity and sophistication of malware, different approaches to performing malware analysis and detection have become popular. Binary-level program analysis techniques are divided into *static analysis*, when the malware is statically analyzed (that is, the program is not executed); and in *dynamic analysis*, when the behavior of the malware program is analyzed during its execution [63]. Malware analysis can help provide information that can be helpful in creating new ways of identifying and remediating malware threats. Unlike static analysis, dynamic analysis can be very costly (time-consuming) as the malware sample runs for a period of time and then its interaction is analyzed. However, static analysis is not without its shortcomings: the binary code of a malware sample can incorporate different techniques to make analysis difficult, such as packaging or code obfuscation, to name a few [48].

Additionally, malware developers have been looking for creative ways to evade detection of their malicious code (especially in dynamic analysis) by adding small pieces of code to their software to detect analysis environments. This is mainly because the longer the malware goes unnoticed, the more revenue cybercriminals earn.

Malicious applications capable of detecting analysis environments are known as *analysis-aware*, *split personality*, or *evasive* malware. Rodríguez et al. [61] and Balzarotti et al. [1] define *analysis-aware* as any malware capable of recognizing when it runs on top of an analysis environment and changing its behavior, avoiding malicious actions that could identify it as malicious software. Polino et al. [57] define evasive malware as malicious applications that target specific analysis techniques through mechanisms such as anti-debugging or anti-instrumentation techniques that allow them to hide their malicious behavior when under analysis or under controlled executions (sandboxed environments). Ekenstein and Norrestam [13] adopt a slightly broader definition of evasive malware, defining it as a malicious application equipped with evasive techniques to prevent malware analysis in a controlled environment. In this document, we adopt the term evasive malware to refer to malware capable of detecting malware analysis environments.

On the other hand, researchers began to use **Dynamic Binary Instrumentation** (**DBI**), a kind of dynamic program analysis technique that allows a security analyst to evaluate a program and verify the actions taken at runtime while modifying its behavior in execution. This technique has been used in several security solutions, such as dynamic taint analysis [66], malware unpacking [35], and VM transparency enhancement [16, 61], among others. More recently, DBI has been used to detect anti-instrumentation techniques in evasive malware. Polino et al. [57] performed an automated analysis of more than $7,000$ samples from malicious applications to find out what anti-instrumentation techniques they use, finding that 15.6% of them used at least one anti-instrumentation technique. In this regard, Kirat et al. [24] suggest taking special care when analyzing evasive malware and propose running them on a full system and other scanning systems, and comparing their observed behaviors. Other authors propose tools such as `PinVMShield` [61] (recently expanded in [15]) and `Arancino` [57] to mitigate specific evasive techniques.

Highlighting the cat and mouse game among security analysts and malware developers, recently, D'Elia et al. [9], Zhechev [76], and Sun et al. [68] started questioning whether DBI-based tools are adequate tools for malware analysis. They claim that it is possible not only to detect DBI-based tools through specific evasion techniques, but also to exploit the attack surface increased by the use of these tools, thus reducing the security of these analysis environments. Remember that an *attack surface* is defined as the subset of system resources used to attack a system, that is, software artifacts related to system vulnerabilities [34]. According to these authors, the reduction in security levels is attributed to the possibility that software flaws that would naturally be difficult to exploit in the real-world become potential vulnerabilities. For instance, some memory zones have write and execute permissions due to the way DBI frameworks work, which can allow malware to gain control flow hijacking very easily, even in situations where pages memory with simultaneous write and execute permissions are prohibited by nature.

In this work, we investigate the evasion of DBI analysis environments through anti-instrumentation techniques commonly incorporated by malicious applications. Our article reviews the most recent research on anti-instrumentation techniques developed to evade DBI analysis and anti-instrumentation countermeasures to overcome these evasion techniques, presenting complete coverage of the state of the art in DBI evasion. Since related literature focuses predominantly on DBI frameworks for desktop platforms, we are limiting the scope of this work to that particular platform.

The contribution of this article is three-fold:

(1) We first review the literature on DBI evasion techniques and countermeasures. Although there are works that describe particular evasion techniques for DBI, to the best of our knowledge, this is the first work that groups them and utilizes a systematic literature review protocol to classify the evasion techniques for DBI. In addition, we also study what countermeasures against evasion techniques are found in the literature.
(2) Then we analyze and compare the taxonomies proposed in the related literature, to propose a new taxonomy that groups all the evasion techniques highlighted in this article. Additionally, we provide a detailed description of the evasion techniques and their countermeasures. In particular, our taxonomy covers 26 types of evasion techniques, while countermeasures for only 12 of them have been found in the related literature.
(3) We finally describe 5 challenges that are areas of interest for future work in the context of DBI framework evasion techniques and countermeasures. These challenges are primarily related to countermeasures, encompassing from the lack of countermeasures, the lack of experimentation in real-world situations, to the lack of impact assessment (in terms of performance overhead and effectiveness) and comparison between them, as well as with the amount of **proof of concepts** (**PoC**), given that only 9 of the 26 evasion techniques present any PoC.

Article Structure. The rest of the article is structured as follows. Section 2 discusses the background to DBI, including its concepts, strengths, and weaknesses, and the main DBI frameworks. Section 3 explains our method of studying the literature. The results of the survey are presented in Section 4, which lists evasion techniques and countermeasures. Section 5 analyzes all the studies presented in the literature review, focusing on which evasion techniques and countermeasures each study considers. Finally, Section 6 brings to the fore the challenges in DBI evasion techniques and opportunities for future research.

2 BACKGROUND

This section first presents the inherent concepts of DBI and then the main frameworks most widely used in the literature for security analysis.

2.1 Dynamic Binary Instrumentation

Nethercote [49] define DBI as analyzing a client application (that is, the program under analysis) while running (at runtime) through instrumentation. The term instrumentation is defined as the act of inserting additional code into a program. Binary analysis refers to the ability to analyze a program at the level of machine code, represented as an object or executable code, while dynamic refers to the ability to do so at runtime. Thus, DBI is an analysis technique that occurs during the execution of the client program, where different code artifacts, known as *analysis code*, are inserted into the client application by an external process (known as *DBI engine*).

DBI has a set of advantages [49]: (i) it does not require any prior preparation in the client program, such as source code recompilation or changes to the client application binary; (ii) naturally covers all the code of the client application during analysis (which is impossible in static analysis, especially when the application generates code dynamically); (iii) and works on binary files and therefore does not require access to the source code of the client application or any prior recompilation. According to Rodríguez et al. [61], other additional advantages are being independent of the underlying programming language and the compiler used to generate the application under analysis and having complete control over the execution of the application under analysis.

Given these advantages, it is not surprising that DBI has attracted a lot of interest, especially in the system security community. While DBI frameworks began to emerge in the early 2000s, today we can find DBI applied in applications related to *taint analysis* [67], *symbolic execution* [75], *cryptoanalysis* [29], *malware analysis* [28, 47], *VM transparency improvement* [61], *automatic unpacking* [35, 57], *vulnerability analysis* [7, 73], *software debugging* [74], *code deobfuscation* [65], *detection of ROP sequences* [8, 69, 70], *software-based fault isolation* [56], and *code randomization* [17], among others.

However, DBI has two main disadvantages. First, the computational cost of the instrumentation drastically impacts execution time. In [60], the performance evaluation of the most common DBI frameworks is studied, reporting experimental results of overheads up to 18x. Second, it has a difficult implementation, since rewriting executable code at runtime can be a hard task. In this regard, Zhechev [76] recently stated that DBI tools may not be appropriate for analyzing potentially malicious code as DBI cannot guarantee, in its default mode, important security properties such as transparency and isolation and thus, we should not trust DBI blindly. In what follows, we present the components of a DBI framework and the two main DBI frameworks that we found to be the target of anti-instrumentation techniques in the literature, Pin [33] and DynamoRIO [4].

2.2 Description of DBI Frameworks

A DBI framework uses a **Just-In-Time** (**JIT**) compiler, which takes as input the program binary code (or application) to be instrumented. Therefore, the framework intercepts the execution of the first instructions of the client application and generates (compiles) a new assembly code directly from the subsequent instructions at runtime. The resulting code is identical to the original, but with modifications determined by the instrumentation framework to ensure that the control over the execution flow is taken and redirected to the analysis code. Also, the new code is allocated in a code cache to eventually reuse it as a way to improve performance.

Today, there is a wide range of DBI frameworks (such as Pin [33], Valgrind [49], FRIDA [51], libdetox [55], DynamoRIO [4], QuarkslaB DBI [59], or DynInst [54], to name a few). They adopt different implementation strategies and have different levels of maturity, as well as different levels of stealth that are constantly improving. Note that achieving full transparency in DBI frameworks in a security sense is very challenging and costly to handle all corner cases, while at the same time this full transparency is required in fewer application domains [4]. DynamoRIO, for example, actively receives updates to mitigate known flaws [2, 30]. In this article, however, we focus only on the two DBI frameworks targeted by evasion techniques in the reviewed literature: in particular, Pin [33] and DynamoRIO [4]. Nevertheless, let us clarify that most of the evasion techniques presented in this articles are directly applicable to other frameworks besides Pin and DynamoRIO, since they are based on the general way of working of DBI frameworks. We briefly explain these two DBI frameworks below.

2.2.1 *Intel Pin.*

The Pin DBI framework [33] (or Pin, for short) allows us to build easy-to-use, portable, transparent, and efficient dynamic instrumentation tools through a set of **Application Programming Interface** (**API**) functions. Designed by Intel in 2005, Pin gives support for the three major desktop **operating systems** (**OS**) (i.e., Windows, Linux, and macOS).

Pin is made up of the three typical components of DBI: (1) the application to be instrumented; (2) a dynamic binary analysis tool developed with Pin, named *pintool*; and (3) the DBI engine, which consists of a **Virtual Machine** (**VM**), a code cache, and an instrumentation API invoked by the pintool. The VM takes as input the native executable code of the application to be instrumented. Code caches are regions of memory controlled by the DBI framework that are used to store instrumented code. The pintool is composed of a mechanism that decides the code to insert and the place of insertion, and the code to execute at insertion points. Starting with version 3, the pintools are compiled utilizing a special runtime, called PinCRT. This change in the Pin framework is motivated to improve the portability of pintools between different compilers and OSs, with the negative side of making it more difficult to migrate pintools developed for version 2, which motivated the creation of a migration guide [22].

Pin can work in two different modes: probe mode and JIT mode. When in probe mode, Pin inserts *probes* (a branching statement is placed at the beginning of the specified routine; also known as *trampoline insertion*) that redirects the control flow to the replacement function. For completing this, it is necessary to relocate the instructions that reside in the insert region. After that, the application and the replacement routine run natively. When in JIT mode, the JIT compiler is responsible for inserting the instrumentation code. The resulting instrumented code is then saved in the code cache, eventually retrieved, and executed. After execution, the JIT compiler obtains the next sequence of instructions to be executed and generates instrumented code again (if necessary). The emulator unit handles instructions that cannot be executed directly, such as system calls that require special handling by the VM [33].

Unlike JIT mode, probe mode does not use code cache. This mode improves performance, but places more responsibility on the tool developer. For example, if a DBI-based tool replaces a function shared by the DBI framework and the instrumented application, undesirable behavior can occur that must be known in advanced and gracefully handled by the developer.

In turn, Pin provides different levels of analysis granularity:

(1) **Instruction:** At the instruction granularity level, Pin allows the pintool to inspect and instrument each assembly instruction at once.
(2) **Trace:** Pin defines a *trace* as a sequence of basic blocks, i.e., a set of assembly instructions that is always entered at the top and can have multiple exits. At the trace granularity level, the instrumentation occurs immediately before a code sequence is executed for the first time.
(3) **Image:** At image granularity level, Pin allows inspection and instrumentation of a complete image code when it is first loaded. In this context, an image represents all data structures for an executable program, including loaded shared libraries. Thus, a pintool can walk through sections in the image, routines in a section, and instructions in a routine.
(4) **Routine:** Routine instrumentation is provided as an alternative to walking through image sections and routines with image-level granularity, as described above. With routine instrumentation, the pintool can inspect and instrument a complete routine after the image is loaded into memory. In Pin, a routine represents any function or procedure of a procedural programming language.

Each level of granularity presented has a set of callback APIs [21] for pintool developers that define the method that Pin will invoke when that type of element is encountered during program execution. For instance, the `INS_AddInstrumentFunction` callback defines which method should be called each time a new instruction is encountered. This design greatly facilitates the development of new analysis tools.

In addition, Pin can instrument child processes of the client application. Pin also intercepts the transfer of control from the kernel to a user-level procedures to maintain control of the application and recover the original interruption context. This kind of control transfer, called *user procedure calls*, occurs, for instance, with asynchronous procedure calls or Windows message callbacks.

Pin is widely used in the scientific community, being noted as the most adopted in related literature [9], in part because it provides efficient instrumentation without shifting the burden to the developer of instrumentation tools [33] and because it provides a very extensive and well-documented API, with many examples. As a result, many interesting pintools have been released. On this matter, it is worth mentioning pintools that allow software debugging for instrumented applications, such as PinADX [32] or the tool introduced in [53].

2.2.2 *DynamoRIO.*

DynamoRIO [4] is a DBI and optimization framework suitable for Windows, Linux, and Android OSs. It also supports IA-32, AMD64, ARM, and AArch64 architectures. Unlike Pin, the source code for DynamoRIO is available under a BSD license.

DynamoRIO works similar to Pin in JIT mode: it enables runtime code transformations in any part of a program, allowing tools to observe and modify the control flow of the application as it runs. DynamoRIO provides complete control over the code stream at runtime through a set of code manipulation APIs. However, it does not have different execution modes like Pin. DynamoRIO runs intermediate to the two Pin modes. In this regard, the code cache and JIT compiler are always used. In fact, the code that resides in the cache is identical to the original, and only necessary changes are made, such as control-flow and other branching instructions, as explicitly required by the instrumentation tool. Unlike Pin, DynamoRIO does not follow a callback-oriented approach.

This DBI framework adopts a set of transparency principles when performing instrumentation to ensure transparency and avoid detection through anti-instrumentation techniques: (i) makes as few modifications as possible to the client application code; (ii) hides any modification made; (iii) and isolates the DBI framework resources, such as its libraries and the heap [3]. However, these adopted principles are insufficient to ensure transparency also in a security sense (that is, the applications under analysis in DynamoRIO are able to recognize the presence of the DBI framework).

3 METHODOLOGY OF THE SYSTEMATIC LITERATURE REVIEW

We follow a systematic methodology to search for relevant research articles and surveys that address evasion and countermeasures techniques in DBI frameworks. We structure this methodology in three steps: *planning*, *conducting*, and *reporting*. These steps are further explained below.

3.1 Planning Step

In this step, we carry out the following activities to establish a review protocol: (1) establishment of the research questions; (2) definition of the search strategy, and (3) selection of studies. Each of them is explained in detail as follows.

Research Questions. The aim of our study is to examine the anti-instrumentation techniques from the following research questions:

Research question 1. What are the anti-instrumentation techniques proposed in the literature?
Research question 2. What are the proposed countermeasures (if any) to mitigate the anti-instrumentation techniques and thus improve the reliability of DBI frameworks?

Search Strategy. We adopt three criteria in the selection of research sources: (i) search for articles in the digital library; (ii) availability of the consulted articles; and (iii) the articles are available in English, in whole or in part. The relevant literature search was conducted in February 2020. The period reviewed includes studies published from 2000 to 2020. We also manually searched journals and books available on the web. For this study, we used

Table 1. Inclusion (IC) and Exclusion (EC) Criteria

#	Criterion
IC1	Articles that discuss evasive techniques applicable to DBI frameworks, malware embedded with these techniques, or countermeasures.
IC2	Articles that discuss concepts of DBI or characteristics of the DBI frameworks, related to evasive techniques.
EC1	Articles in which the language is different from English or Spanish cannot be selected.
EC2	Articles that are not available for reading and data collection (articles that are only accessible through pay-walls or are not provided by the search engine) cannot be selected.
EC3	Duplicate articles cannot be selected.
EC4	Publications that do not meet any of the inclusion criteria cannot be selected.

three research databases: (i) ACM Digital Library[1]; (ii) Science Direct[2]; (iii) SpringerLink[3]; and (iv) IEEEXplore Digital Library.[4] One of the reasons that motivated us to use these databases is because they index several top-notch conferences in the areas of system security and malware analysis (such as IEEE S&P, RAID, and ACSAC, just to name a few of them). We use a search string composed of three concepts: *DBI*, *evasion*, and *malware*, considering also their alternative terms and synonyms, such as"DBI" or DBI; and "evasion", "analysis-aware", or "split-personality". Apart from these databases, we have manually scrutinized the DBLP website of the top-notch conferences which are not indexed in these search databases, such as NDSS and USENIX Security.

Selection of Primary Studies. This activity comprises two steps. In the first step, called *1st filter*, we evaluated only the title and the abstract of each article according to the inclusion and exclusion criteria (see Table 1) and selected the articles within the scope of the research questions. In the second stage (*2nd filter*), we carefully read the selected works and finally included them if they presented any evasion technique and/or countermeasure (regardless of whether a PoC is provided).

3.2 Conducting Step

The application of the review protocol yielded the following preliminary results. A total of 483 papers were returned from the search string, 391 from ACM, 91 from Science Direct, 6 from SpringerLink, and only 1 from IEEEXplore. In the first filter, we selected 57 articles by analyzing the abstracts and eliminating duplicates. In the second filter, the full-text analysis was performed and as a result several of these articles were discarded. Most of these articles address different anti-analysis techniques used by malware, such as anti-debug and anti-VM, which is a broad field of research but is beyond the scope of this article. Some other articles were discarded due to lack of connection with our research questions. In addition, we discarded articles that propose countermeasures or tools that are not intended to mitigate anti-instrumentation techniques, at least as a secondary objective.

After this stage, seven articles remained. We then snowball search these articles, that is, we individually retrieved the articles referenced in the selected articles, as well as other works that reference them. These articles and works were examined analogously and added to the selected set when they deemed relevant. In the end, we obtained 10 more artifacts: 5 articles, 4 gray research papers (presentations at recognized industry conferences such as BlackHat USA and BlackHat Asia), and 1 tool, which were analyzed in relation to some evasion technique and/or countermeasure. The next section looks at our findings in detail.

[1]Available through the web address https://dl.acm.org/search/advanced.
[2]Available through the web address https://www.sciencedirect.com/.
[3]Available through the web address https://link.springer.com/.
[4]Available through the web address https://ieeexplore.ieee.org/Xplore/home.jsp.

4 LITERATURE REVIEW

Based on the results of the systematic review, this section addresses both the evasion techniques for DBI frameworks and the protections (countermeasures) against these techniques. To improve readability, we decided to divide this section into three parts. We first discuss the existing classifications of DBI evasion techniques and introduce a new taxonomy. Then we explain the evasion techniques in DBI and finally the existing countermeasures.

4.1 Towards a New Taxonomy of DBI Evasive Techniques

We found six articles that propose taxonomies for DBI evasive techniques. In the sequel, we review these works in chronological order and discuss them in detail, proposing a new taxonomy that unifies all previous taxonomies.

The first classification of techniques to detect DBI frames is proposed by Rodríguez et al. in [61]. Specifically, the classification focuses on anti-instrumentation techniques targeting Pin. The authors classify DBI evasion techniques into *indirect*, when the evasion technique does not incorporate any code artifacts to detect the DBI framework, and *direct* otherwise.

Indirect evasion techniques fall into two categories:

(1) **Time-based**, which groups evasion techniques that are not intended to detect the scanning environment itself, but rather to postpone the execution of malicious behavior or execute it at a specific date and time.
(2) **Event-based**, which groups techniques that condition the execution of malicious behaviors to the occurrence of specific user or system events.

Direct evasion techniques are divided into six categories:

(1) **Pin VM detection**, which groups evasion techniques that focus on recognizing the presence of the shared dynamic library associated with the Pin VM (specifically, `pinvm.dll`).
(2) **Communication channel detection**, which contains the set of anti-instrumentation techniques that detect the **inter-process communication** (**IPC**) mechanisms used by Pin.
(3) **Time variation detection**, which groups the techniques that aim to detect the presence of Pin through variations in the execution time.
(4) **Pin JIT compiler detection**, which contains the set of evasion techniques that identify the presence of the Pin JIT compiler through artifacts left in the memory address space of the client application process.
(5) **Detection by instruction pointer (IP) register**, which groups the techniques that detect the presence of Pin by monitoring and validating the value of the IP register during the execution of the application under analysis.
(6) **Detection by other techniques (miscellaneous)**, which adds evasion techniques that do not fit into the other categories and use the existence of other execution artifacts to reveal the presence of Pin.

Sun et al. [68] propose a classification where the evasion techniques for the Pin and DynamoRIO frameworks are grouped into two main categories:

(1) **Passive gate-coded detections**, which includes anti-instrumentation techniques that allow bypassing the DBI analysis without the need to detect them previously.
(2) **Active detection**, which includes techniques that focus on actively detecting instrumentation frames and then evading them.

In Polino et al. [57], DBI evasion techniques fall into four categories:

(1) **Code cache artifacts**, which considers techniques that detect when a code snippet is executing from a memory region that belongs to a code cache.

(2) **Environment artifacts**, which groups the techniques that allow the detection of DBI frameworks using memory and system artifacts left by the instrumentation process during its execution.
(3) **JIT compiler detection**, which contemplates the techniques aimed at detecting the instrumentation framework's JIT compiler through its interactions with the system.
(4) **Overhead detection**, which includes techniques that determine when a code snippet is under instrumentation based on past performance metrics of the client application.

The works of Kirsch et al. [25] and Zhechev [76] group evasion techniques into three categories, according to the specific component of the DBI framework that is being attacked and exploited:

(1) **Code cache**, which groups anti-instrumentation techniques that detect anomalies in memory introduced by the use of code caches.
(2) **JIT compiler overload**, which includes the set of evasion techniques that aim to detect the time overload introduced by the use of the JIT compiler.
(3) **Runtime environment artifacts**, which understands techniques that look for environment artifacts introduced by DBI frameworks when analyzing a client application.

Finally, D'Elia et al. [9] propose a classification of evasive techniques into seven categories, from the point of view of an adversary:

(1) **Time overhead**, where an adversary uses generic timing strategies to compare the execution of a code snippet to a benchmark and look for significant discrepancies.
(2) **Leaked code pointers**, where an adversary breaks the DBI transparency property, identifying the actual IP.
(3) **Memory content and permissions**, where an adversary inspects the memory address space to reveal additional artifacts such as unexpected exported functions or sections that belong to the runtime of the DBI framework.
(4) **DBI engine internals**, where changes applied to the execution context by DBI frameworks, such as modifications to shared libraries, are checked and used to detect the DBI framework.
(5) **Interactions with the** OS, where an adversary detects the presence of DBI frameworks when interacting with the OS. For instance, checking the parent process or the list of active processes.
(6) **Exception handling**, where an adversary deliberately raises exceptions to check how they are handled. This anti-analysis technique is also commonly used by malware to detect the presence of debugging tools.
(7) **Translation defects**, where an adversary scans for implementation flaws and limitations, such as unimplemented or unsupported assembly instructions, to discover the presence of DBI frameworks.

Discussion on existing taxonomies. The classifications proposed in [61] and [68] are very similar. The category of direct evasion techniques in [61] is equivalent to the category of active detection techniques proposed in [68], bringing together evasion techniques focused on first detecting the DBI framework and then evading it.

Similarly, the indirect evasion of [61] and the passive gate-coded detection techniques in [68] are equivalent, comprising the techniques that evade instrumentation frameworks without detecting them.

Likewise, the taxonomy proposed in [61] divides the category of direct evasion techniques into six subcategories. These subdivisions are similar to the classifications proposed in [9, 25, 57, 76], grouping evasion techniques under similar criteria.

In particular, the categories *Pin VM detection*, *Communication channels detection*, and *Detection by other techniques (miscellaneous)* of [61] are contained in the categories *Environment artifacts* and *JIT compiler detection* of [57], in *Runtime environment artifacts* of [25, 76] and also in *DBI engine internals* and *Interactions with the OS* of [9].

The category *Detection by IP register* of [61] and *Leaked code pointers* of [9] are equivalent as well. In addition, both are contained in the categories *Code cache artifacts* of [57] and *Code cache* of [25, 76]. Finally, the categories *Time variation detection* of [61], *Overhead detection* of [57], *JIT compiler overhead* of [25, 76] and *Time overhead* of [9] are also equivalent.

Despite these similarities, the classification proposed in [61] aims to group evasion techniques focused exclusively on the Pin instrumentation framework, while the classification in [68] focuses on Pin and DynamoRIO. Unlike these, the classifications of [9, 25, 57, 76] are more general classifications (that is, they are independent of a specific DBI framework). However, these other taxonomies do not consider indirect evasion techniques, which makes their taxonomies unable to group a series of anti-instrumentation techniques that are successfully covered by [61] and [68].

Considering all this, in this article we propose a new taxonomy that is more general, since it takes anti-instrumentation techniques into account independently of the DBI framework and contemplates the direct and indirect nature of anti-instrumentation techniques. We have adopted the division into direct and indirect evasion techniques initially proposed in [61], but dividing the direct evasion techniques as proposed in [57]. Regarding indirect techniques, we divided them into two new classes based on the limitation that each evasive technique exploits. Recall that direct evasion techniques focus first on detecting the presence of a DBI framework and then circumventing it, whereas indirect evasion techniques do not incorporate any code artifacts to detect a DBI framework and evasion occurs without having to detect it. Figure 1 illustrates the classification resulting from anti-instrumentation techniques. Each technique is described in more detail below.

4.2 Description of Anti-Instrumentation Techniques

4.2.1 Indirect Anti-Instrumentation Techniques. Indirect anti-instrumentation techniques are those that lead to the evasion of DBI frameworks without the need to detect them. We can divide this class into techniques that explore the *functional limitations* of DBI frameworks and techniques that exploit the *resource limitations* of analysis environments.

Functional Limitation. Analysis environments typically need to be able to reproduce standard behaviors from bare metal, OSs, and other environments to perform application analysis transparently. However, not all possible behaviors are always handled and implemented in these environments, eventually leading to behavioral inconsistencies between bare-metal systems and analysis systems. These differences can result in a change in the behavior of the analyzed program or even an attempt to hamper the analysis system. We identify two types of functional limitations in DBI frameworks: *unsupported instructions* and *unsupported behaviors*.

Despite the similar names, unsupported instructions and unsupported behaviors are different evasive techniques. Unsupported instructions are instructions that belong to the underlying instruction set architecture that are not handled correctly by the DBI framework, resulting in termination of the application under analysis with graceful exit due to incompatibility issues generating a notification to the user, while the unsupported behaviors cause an abrupt termination of the application under analysis and the DBI framework due to an unhandled exception.

Evasion Technique 1. Unsupported Assembly Instructions. DBI frameworks need to handle all the instructions available in an underlying architecture, as any of these instructions are likely to be found in the client application code. However, Sun et al. [68] demonstrate that certain DBI frameworks fail in this regard. In Pin, for example, executing the `retf` instruction[5] by an application under analysis results in the termination of the application and Pin with the message *"Pin doesn't support FAR RET [...]"*, which makes the DBI tool unable to analyze the application. Unfortunately, the documentation for DBI frameworks lacks information on operational

[5]A return to a caller procedure located in a code segment other than the current one [19].

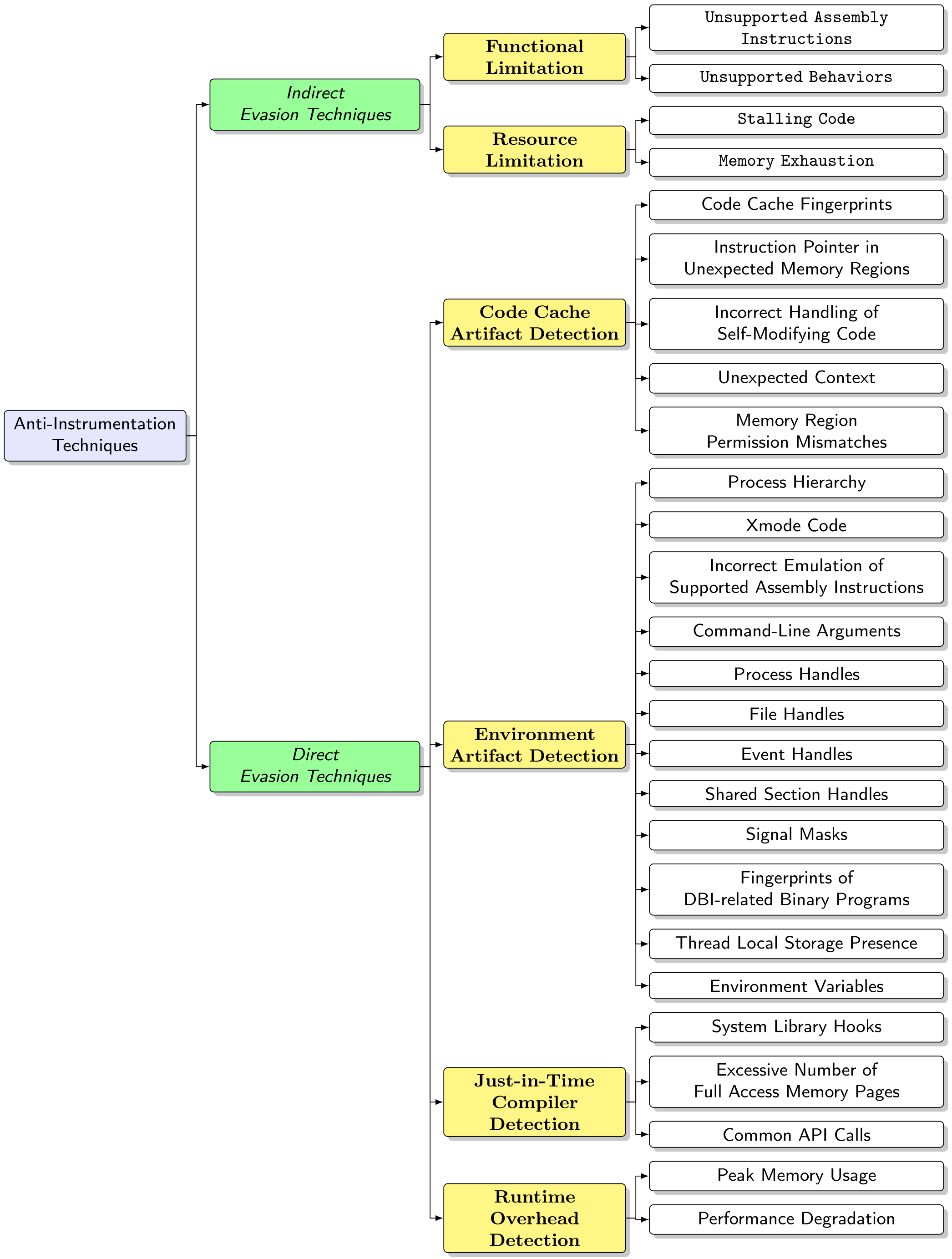
Anti-Instrumentation Techniques
Indirect Evasion Techniques
Functional Limitation
Unsupported Assembly Instructions
Unsupported Behaviors
Resource Limitation
Stalling Code
Memory Exhaustion
Direct Evasion Techniques
Code Cache Artifact Detection
Code Cache Fingerprints
Instruction Pointer in Unexpected Memory Regions
Incorrect Handling of Self-Modifying Code
Unexpected Context
Memory Region Permission Mismatches
Environment Artifact Detection
Process Hierarchy
Xmode Code
Incorrect Emulation of Supported Assembly Instructions
Command-Line Arguments
Process Handles
File Handles
Event Handles
Shared Section Handles
Signal Masks
Fingerprints of DBI-related Binary Programs
Thread Local Storage Presence
Environment Variables
Just-in-Time Compiler Detection
System Library Hooks
Excessive Number of Full Access Memory Pages
Common API Calls
Runtime Overhead Detection
Peak Memory Usage
Performance Degradation

Fig. 1. Proposed taxonomy for anti-instrumentation techniques.

limitations. Another case of unsupported instruction occurs with the Intel IA-32 `enter` instruction in the Valgrind DBI framework [50], which cannot continue as it does not implement a specific handler for this instruction.

Evasion Technique 2. Unsupported Behaviors. Certain client application code snippets may run correctly when run in native mode, but cause exceptions or abrupt terminations when run on DBI frameworks since what they do is not supported. Li and Li [30] are the first authors to suggest the use of these code snippets as an evasive technique. In particular, the authors suggest using the heap as the stack (an unconventional programming practice) as an evasion example for DynamoRIO. Another example is the multi-threaded handling of Pin version 2. According to [11], Pin can intercept threads created by remote processes and instrument them appropriately. However, as noted in [72], this version of Pin does not support injection into processes that have more than one active thread.

Resource Limitation. Generally, analysis environments have limited processing resources. Therefore, the characteristics of the environment (such as the number of processors, the number of cores, or the amount of memory available) and the characteristics of analysis (such as the time taken to run and observe the execution of a malware sample) are limited. Common anti-instrumentation techniques that take advantage of this limitation are the stalling code and memory exhaustion.

Evasion Technique 3. Stalling Code. Stalling code is an evasion technique that does not aim to detect the analysis environment but rather to postpone the execution of malicious behaviors, taking advantage of the time limitation of the analysis tools. Therefore, this technique is effective against all dynamic analysis tools in general. Be aware that any analysis environment with a run-time limit is potentially vulnerable to this evasion technique. Rodríguez et al. [61] suggest its use to evade the analysis in Pin.

Kolbitsch et al. [26] define the stalling code as a sequence of instructions that fulfills two properties: (i) it executes considerably slower in the analysis environment than in native mode (that is, it causes runtime overhead); and (ii) the total execution time of the stalling code must be "not negligible" with regard to the analysis time limit imposed by the analysis environment. According to the authors, the blocking code is typically implemented through a loop containing "slow" operations, such as system calls.

Evasion Technique 4. Memory Exhaustion. Memory exhaustion attacks are a type of resource exhaustion attacks in which the adversary occupies a significant amount within the system and thus enabling a denial of service in the application.

Although it is not considered in any taxonomy in the literature, the idea was originally proposed by D'Elia et al. [9]. Since a DBI framework's code cache will take up some memory space that would otherwise be available, a sophisticated adversary can take advantage of this and push the limits of memory allocations to indirectly discover the presence of a DBI framework.

4.2.2 Direct Anti-Instrumentation Techniques. As explained above, direct evasion techniques focus first on detecting the presence of a DBI framework and then circumventing it. These techniques include code cache artifact detection, environment artifact detection, JIT compiler detection, and runtime overhead detection.

Code Cache Artifact Detection. These anti-instrumentation techniques exploit particular artifacts and behaviors that DBI frameworks use in code caches. In general, the techniques grouped in this category take advantage of the fact that the code that is running while being analyzed through DBI is not the original code of the client application. In what follows, we discuss different detection techniques of this type.

Evasion Technique 5. Code Cache Fingerprints. The DBI framework maintains an instrumented version of the application code being analyzed in the code cache. Sun et al. [68] show that this copy can be found by scanning the memory of the process under analysis. Therefore, if the client application detects two or more occurrences of a code snippet that should be unique, this would indicate the presence of the DBI framework. Also, determining the memory address of code caches can lead to the instrumentation framework escaping, as shown in [9, 25, 68, 76]. The DBI framework escape is formally described in [9] as "*(...) when parts of its instruction and data flow get out of the DBI engine's control and execute natively on the CPU, in the address space of the analysis processor of a different one.*"

Sun et al. [68] suggest that code caches can be detected by identifying memory artifacts (such as signatures) that are characteristic of the code caches of DBI frameworks. An example of such signatures is the hexadecimal string `0xfeedbeaf`, which is present in memory regions that Pin uses as code caches. Later, Polino et al. [57] suggest, analogously to Sun et al. [68], that DBI frameworks can be detected by scanning the memory address space of the process under analysis for artifacts related to DBI frameworks, such as certain specific strings and code patterns.

Evasion Technique 6. IP in Unexpected Memory Regions. When an application is being analyzed by a DBI framework, the running code comes from memory regions known as code caches, rather than the memory regions to which the executable was originally mapped. This difference in memory regions can be observed through the value of the IP register, the logical CPU register responsible for storing the memory address of the next assembly instruction to be executed.

Because DBI frameworks use code caches, the original binary code of the instrumented application does not run. Thus, several works [9, 14, 18, 25, 57, 61, 76] suggest that a client application can detect that the code that is being executed is not in the same memory region as the original application code through of the IP register value. Various mechanisms can be used to inspect the IP register value, such as the Intel assembly instructions **`fstenv`**, **`fsave`**, and **`fxsave`**. Note that the ASLR relocation process can produce a false positive detection in some situations.

*Evasion Technique 7. Incorrect Handling of **Self-Modifying Code (SMC)**.* As discussed above, since the original application code does not run, but rather its (instrumented) copy resides in code caches, the DBI framework needs to handle the SMC appropriately. An SMC is code that modifies its instructions by overwriting them at runtime [57].

In general, DBI frameworks can handle SMC correctly when the specific instructions being modified are not in the code cache. However, according to Hron and Jermar [18] and Polino et al. [57], if the SMC is not handled correctly, the client application may detect the presence of the analysis tool or have runtime errors that can lead to an abrupt termination, making the analysis impossible in both cases.

Evasion Technique 8. Unexpected Context. Sun et al. [68] showed that Pin maintains a copy of all CPU logical registers in a code cache region used as a backup. At runtime, the `EBX` register is reserved for Pin use only as it maintains the address of this memory region. When the client application code uses the `EBX` register, Pin takes responsibility for rewriting it to use another record. Based on this behavior, the authors demonstrate that if the client application finds the backup copies of the records in memory, it is possible to change the copy of the `EBX` register used by Pin (thus disabling the analysis), as well as reveal the presence of this DBI framework.

Evasion Technique 9. Memory Region Permission Mismatches. This category includes techniques for detecting discrepancies in the permissions assigned to memory regions. Two evasion techniques of this type are proposed to detect the presence of Pin and DynamoRIO frameworks in [18]: (1) block cache vs. virtual memory state, and (2) inherent weakness of the write-protecting approach in memory regions. The first technique uses the fact that code caches always have read, write, and execute permissions. If the client code tries to change the memory permissions of the region where the code resides setting a guard page protection [46], the DBI framework must ensure that the permissions for the application have been changed accordingly. Otherwise, the application can detect the DBI framework. The second technique takes advantage of the exception handling behavior of write attempts to a memory region without proper permissions. When writing to a region marked as write-protected, an exception is thrown that leads the flow of execution to the corresponding exception handling mechanism. In other words, the application under analysis marks a region as write-protected, deliberately writes to it to cause the exception, and inspects the stack, comparing it to previous states. The DBI framework must ensure that the data that remains on the stack when the exception handler runs does not reveal its presence.

No-eXecute Bit (**NX**) is a technology developed by processor manufacturers AMD and Intel that allows OSs to mark writable memory pages as non-executable as a security mechanism. This technology has been named differently by manufacturers. For example, AMD maintains the NX terminology, while Intel named it ***eXecution Disable*** (**XD**) and ARM as ***eXecute Never*** (**XN**). In software, this technology is known as W $\oplus$ X or as Data Execution Prevention due to the wide acceptance of Windows terminology.

According to Kirsch et al. [25] and Zhechev [76], the DBI Pin and DynamoRIO frameworks neglect this marking bit. So what happens in practice is that memory regions that cannot be run natively can be executed when an application is under the analysis of a DBI tool, since the code is executed in a code cache region of the DBI framework, which is a memory region with write and execute permissions. Therefore, the authors proved that DBI framework detection is feasible by testing whether W$\oplus$ X is active or not. Note that the techniques in this category are different from the *SMC* techniques described above, as these techniques abuse unexpected mismatches in memory regions, while *SMC* techniques refer to the existence of program code capable of natively modifying itself that is incorrectly handled by the DBI framework.

Environment Artifact Detection. Environment artifacts are the most commonly used information to detect the presence of DBI frameworks. Of the 26 evasion techniques discussed in this article, 12 of them focus on environmental artifacts, since DBI frameworks imply important modifications to the process of the client application, from changes in the memory space of the process to changes in the synchronization mechanisms and thread communication. We describe these techniques in detail below.

Evasion Technique 10. Process Hierarchy. When an application is instrumented, it runs as a child process of the DBI framework. Therefore, the client application can check the name of its parent process to determine if it is a DBI tool. This detection technique is initially proposed by Falcón et al. [14] to detect Pin. Li and Li [30] also demonstrate their efficacy against DynamoRIO. Several works [14, 30, 57, 61] suggest using this technique to detect and subsequently evade DBI frameworks.

Evasion Technique 11. Xmode Code. Xmode Code's evasion technique exploits the subsystems of 64-bit environments that provide support for 32-bit applications, as Windows does with WoW64 ("Windows 32-bit on Windows 64-bit"). WoW64 is an emulator mode that allows a program to run transparent 32-bit code in 64-bit environments [42]. When a program designed for 64-bit systems is running, the CS register has the hexadecimal value `0x33`, indicating that the code is running in 64-bit mode. In contrast, when a 32-bit code application runs on a 64-bit system via 32-bit compatibility mode, the CS register has a hexadecimal value of `0x23`, indicating that it is in compatibility mode.

However, according to Sun et al. [68], the CS register is always set to compatibility mode and cannot switch to 64-bit mode when an application is under DBI analysis. The client application can use this behavior to detect the presence of the DBI framework. When running 64-bit code incompatible with 32-bit mode and the CS register is in compatibility mode, an abrupt termination may occur. This technique has been shown to work against the Pin and DynamoRIO frameworks. The authors demonstrated that while in Pin the CS register is always set to compatibility mode, attempting to change it in DynamoRIO results in an application crash.

Evasion Technique 12. Incorrect Emulation of Supported Assembly Instructions. Recall that DBI frameworks need to handle all the instructions available in an underlying architecture. However, some of the supported instructions are incorrectly emulated, and these discrepancies allow the presence of a DBI framework to be detected. Note that this technique is different from *Unsupported Instructions* in that the instructions are supported, but are not emulated correctly.

For instance, Kirsch et al. [25] and Zhechev [76] showed that Pin has a flaw in handling system calls in Linux environments. Without DBI, when a system call is executed through the `syscall` command, the RCX CPU logical register is overwritten with the contents of the IP register, allowing the OS to resume normal execution program after ending the system call. However, since the DBI framework wraps all system calls made by the instrumented application and executes them in other contexts, the RCX registry is not modified accordingly. Therefore, this discrepancy can be used as a detection mechanism.

Likewise, the assembly instruction **`rdfsbase`**, available on newer Intel processors (Ivy Bridge and later), allows the program to save the contents of the FS and GS segment registers to a target register. Pin cannot emulate this instruction correctly, though, and instead saves the segment records from the DBI framework. Therefore, an application can detect if it is being instrumented by the `SYS_arch_prctl` system call, which also returns the content of the FS and GS segment registers, and comparing it with the result obtained from the execution of the **`rdfsbase`** instruction. This technique has been successfully tested in Linux environments by Kirsch et al. [25] and Zhechev [76]. However, as the author explain, the applicability of this technique in real-world scenarios is reduced since the OS does not allow user-mode code to execute this instruction by default, since it is a privileged instruction.

Evasion Technique 13. Command-Line Arguments. Some strings related to a DBI framework like Pin can be identified by checking the command line arguments passed by the main process of an application under DBI parsing [14]. According to Microsoft documentation [44], command-line arguments are passed to the program as an array of null-terminated strings representing each user-supplied argument at launch.

DBI frameworks, like Pin, use command-line arguments at launch to define the application to be analyzed and the analysis tool to use. Since the client application shares memory address space with the DBI framework, it is possible to scan the entire memory space for command-line arguments to determine if the client application is under instrumentation. Rodríguez et al. [61] corroborate this technique together with Falcón and Riva [14].

Evasion Technique 14. Process Handles. A process handle is a unique handle that is generated when a new process is created in Windows [41]. Since the client application process is created by the DBI framework, it maintains an active process handle for the client application. According to Falcón and Riva [14], when a Windows application is under Pin, its presence is detectable by enumerating the handles of active processes.

Evasion Technique 15. File Handles. File handles are unique identifiers associated with a file when it is opened and assigned by Windows [39]. When a program is analyzed by a DBI framework, the number of active file handles is typically greater than when it is run independently. In this regard, Li and Li [30] show that the maximum number of file handles that can be used in a process is less when the DynamoRIO framework is running compared to running the native application. In particular, they demonstrated a reduction of `4,096` file handles to `4,000`. In addition, an application has a maximum number of file handles that can be used at the same time. Some OSs allow the user to change the maximum number of file handles. In this regard, they also demonstrate that the maximum number of file handles that a process can use cannot be changed during execution when running on the DBI DynamoRIO framework. On Linux, a common way to change this maximum value is by using the `setrlimit` system call. However, according to the authors, this system call always fails, regardless of the maximum number of file handles that are set. Therefore, an application can detect the presence of the instrumentation framework if it fails to change the maximum number of file handles at runtime.

Evasion Technique 16. Event Handles. Event objects are part of Windows IPC mechanisms that are useful for sending signals to threads in a process (or even other processes) by reporting the occurrence of a particular event [38]. Once an event object is created, it is handled by an event handler. According to Falcón and Riva [14], Pin makes extensive use of IPC and thus, the client application can enumerate and verify the existing event handlers to reveal the presence of Pin. Rodríguez et al. [61] corroborate with [14], confirming the effectiveness of this technique and stating that the IPC channel names used by Pin usually have the prefix "*Pin_IPC*".

Evasion Technique 17. Shared Section Handles. Shared sections is a feature provided by Windows that is commonly used for IPC [6] and allows sharing portions of virtual memory between processes. Once a shared section is created, it is manipulated through identifiers. Falcón and Riva [14] demonstrate that Pin makes use of this functionality and is therefore suitable for DBI framework detection.

Evasion Technique 18. Signal Masks. On Linux-based OSs, it is possible to verify which signals are being monitored by a process using the mask `SigCgt` [31]. Li and Li [30] demonstrate that DynamoRIO monitors all signals while avoiding modifying them through this mask and therefore an application can detect the presence of DynamoRIO by checking the mask `SigCgt` assigned to its process. The literature reviewed does not point to successful detections of other DBI frameworks using this technique.

Evasion Technique 19. Fingerprints of DBI-related Binary Programs. DBI frameworks commonly inject their analysis libraries into the process under analysis. Pin, for example, injects the `pinvm.dll` library, which can be detected by scanning the virtual memory of the process for string patterns or even looking for the functions exported by all libraries to find `PinWinMain` and `CharmVersionC` functions, as shown by Falcón and Riva [14]. Similarly, the same authors also propose to recognize common functions exported by all pintools to detect Pin. Furthermore, this technique can also be extended to detect DynamoRIO, as shown by Li and Li [30].

Likewise, when an application is under analysis in Pin, there are code sections related to the DBI framework allocated in the virtual memory space of the application process. Falcón and Riva [14] show that they are sections related to pintools and the `pinvm.dll` library (such as `.pinclie` or `.charmve`, to name a few). Therefore, a client application can scan its memory space to detect these sections and thus reveal the presence of Pin.

Evasion Technique 20. ***Thread Local Storage (TLS)****Presence.* According to the Microsoft manual [43], TLS is a feature provided by Windows that allows unique data to be provided for each thread of a process stored in a vector data structure, with a minimum of 64 positions and a maximum of 1, 088, accessible through a global index. Therefore, one thread allocates an index that can be used by other threads to retrieve the unique data associated with that position in the vector. That is, TLS variables can be seen as global variables visible only to a particular thread and not in the entire program.

According to Sun et al. [68], Pin makes use of TLS and this behavior can be observed from the client application. Consequently, an application under Pin analysis can check the TLS indices that are used, revealing the presence of the positions used by Pin.

Evasion Technique 21. Environment Variables. Environment variables become visible (and editable) to child processes. As discussed above, the client application runs as a child process of the DBI framework. According to Kirsch et al. [25] and Zhechev [76], certain environment variables are necessary for the correct execution of Pin in Linux, such as `PIN_INJECTOR64_LD_LIBRARY_PATH`. Therefore, the client application can access these variables at runtime and thus detect Pin.

JIT Compiler Detection. The DBI frameworks have as one of the main components the JIT compiler, which has as input the executable of the application to be analyzed and as output the instrumented instructions that will eventually be executed. However, as Polino et al. [57] point out, the JIT compilers of DBI frameworks generate a lot of activity within the process being analyzed. For example, each time the code generated by the JIT compiler must be allocated to a code cache, a large number of system calls are required to access memory and allocate memory pages properly. Next, we present techniques that allow us to detect the effects of JIT compiler actions on the process under DBI analysis.

Evasion Technique 22. System Library Hooks. When Pin runs, some system libraries are modified with detours (hooks) in certain functions provided by those libraries. In this way, Pin can intercept the system call instructions at a point where the system call number and arguments are captured, and then transfer control to the VM monitor [64]. As indicated in [64] and later discovered empirically by Falcón and Riva [14], one of these libraries is `ntdll.dll`, which is responsible for exporting the native OS API [12]. As indicated in Rodríguez et al. [61], the Pin framework JIT compiler is responsible for making these changes. Polino et al. [57] further state that these hooks can always be found at the beginning of modified functions. Therefore, an application under the analysis of a DBI tool can inspect the initial instructions of the OS routines that are usually modified by DBI frameworks and thus detect them.

Evasion Technique 23. Excessive Number of Full Access Memory Pages. As shown by Falcón and Riva [14], the JIT compiler of a DBI framework makes heavy use of memory pages with read, write, and execute permissions. Therefore, a client application can use the number of these pages in the virtual memory of a process as a criterion for indicating the presence of the Pin framework. Later, Rodríguez et al. [61] and Polino et al. [57] confirmed the effectiveness of this technique to detect Pin. Sun et al. [68] also demonstrated that memory page permissions can be used in conjunction with other evasion techniques, such as code cache detection, to detect DBI frameworks.

However, both Falcón and Riva [14] and Rodríguez et al. [61] warn that this detection technique can lead to false positives, since the application being analyzed may have a JIT compiler or allocate a large number of memory pages with similar permissions.

Evasion Technique 24. Common API Calls. To create and manage memory pages used during the instrumentation process, Pin makes heavy use of a set of OS routines (APIs), such as `ZwAllocateVirtualMemory` [14, 57] among others. Falcón and Riva [14] demonstrate that an application can monitor the use of these APIs to detect the DBI tool.

Polino et al. [57] suggest that a client application can monitor the number of calls to these routines by adding multiple execution counters. In particular, the application can insert a sequence of assembly instructions at the beginning of the API that is monitored to increment a variable counter each time the function is called. In this way, the variable counter allows to determine the number of calls to that API. Therefore, it would be possible to compare the number of calls made with the threshold of the expected number of calls. However, as before, Falcón and Riva [14] warn that using the frequency of use of certain APIs as the sole criteria for signaling the presence of DBI frames can also lead to false positives.

Runtime Overhead Detection. This category includes techniques based on taking some measurements at runtime and then comparing them with a baseline, which is generally made up of measurements taken in advance by the client application when it runs natively. DBI frameworks analyze a client application after instrumenting its code appropriately, considering the analysis code. This parsing activity takes time that causes a measurable degradation from expected performance. Likewise, the amount of memory used by the instrumented application is greater than when it runs natively. Discrepancies in these measurements with regard to baseline measurements serve as clues for the client application to detect the presence of a foreign agent, such as a DBI framework. Common techniques that take advantage of these measurable differences are *peak memory usage* and *run-time discrepancies.*

Evasion Technique 25. Peak Memory Usage. Li and Li [30] show that the peak of virtual memory used by an application is higher when the process is under the analysis of the DynamoRIO framework. This increase is due to artifacts inserted by the DBI framework into the process's address space, such as code caches and the DynamoRIO library. Thus, if an application can compare the memory peaks used under instrumentation and those expected in native execution, it is possible to detect the presence of the DBI framework.

Evasion Technique 26. Performance Degradation. DBI frameworks are known to insert additional computational cost into client application execution, observable as runtime overhead [33, 49]. Falcón and Riva [14] show that this difference in execution time can be used as a criterion to detect Pin. Therefore, an application under DBI analysis can calculate the elapsed time to execute a code snippet and compare this value to a threshold associated with the expected execution of that snippet. If the elapsed time is greater than the threshold, this indicates the presence of the DBI frame. Additionally, they suggest that a client application can deliberately perform behaviors that are more computationally costly when a DBI framework is present (such as loading libraries) to increase runtime overhead and thus reduce the number of false positives. Rodríguez et al. [61] corroborate with [14] the effectiveness of this technique. However, both of them also warn of possible false positives, as the application runtime can vary with regard to the environment.

Polino et al. [57] list specific mechanisms, such as assembly instructions and Windows APIs, that an application can use to obtain the elapsed execution time of an application and then reveal the presence of a DBI framework. In this regard, Kirsch et al. [25] and Zhechev [76] present two cases in which the overhead imposed by the JIT compiler is more evident: in the interactions of the execution in the first iteration of a loop and in the loading and unloading of the same library.

4.3 Description of Countermeasures against Anti-Instrumentation Techniques

In this section, we present the countermeasures used to avoid the evasive techniques of the DBI framework. We first introduce tools that implement these countermeasures and then explain each countermeasure in more detail.

4.3.1 Description of Tools. We found three anti-instrumentation tools (`PinVMShield` [15, 61], `Arancino` [57], and an unnamed library [9]) during our systematic review that implement countermeasures to some of the DBI framework detection techniques described above. Note that not all the techniques presented above are currently mitigated by these tools and furthermore many of the proposed mitigation mechanisms are focused on a specific DBI framework and therefore may not be extensible to other DBI frameworks.

One of the first tools in the academic literature that detects and bypasses the evasion techniques used by evasive malware, including some Pin anti-DBI techniques, is `PinVMShield`, introduced by Rodríguez et al. in [61]. The tool, developed for Windows, follows a plug-in architecture, allowing the extension to cover new evasion techniques. It is released under the GNU/GPL version 3 license and its source code is publicly available at https://bitbucket.org/rjrodriguez/pinvmshield/. This tool has recently been expanded in [15]. Similarly, Polino et al. [57] propose a set of countermeasures for certain anti-instrumentation techniques targeting Windows and implemented these protections on Pin. The resulting tool was named `Arancino`, and its code is also publicly available at https://github.com/necst/arancino. Recently, D'Elia et al. [9] proposed a DBI library to detect and mitigate DBI evasion and instrumentation escape. The tool targets to Pin running on Windows and its source code is also publicly available at https://github.com/season-lab/sok-dbi-security/.

4.3.2 Description of Countermeasures. The tools mentioned above focus on a set of countermeasures that address specific anti-DBI techniques. In the following, each of these countermeasures is explained according to the evasive technique that is mitigated.

Countermeasure Against *Code Cache Fingerprinting, Unexpected Context, Excessive Number of Full Access Memory Pages,* *and* ***Memory Region Permission Mismatches.*** All of these evasion techniques are based on the inspection of memory artifacts. In this regard, `Arancino` [57] monitors certain system functions (such as `VirtualQuery` and `NtQueryVirtualMemory`) that allow a program to query and interact with its own memory space process virtual, detecting queries for memory artifacts related to DBI frameworks. In particular, the memory addresses that are queried are checked to see if the client application can access those addresses. Valid memory addresses are determined by an allow-list policy, created at the beginning of the instrumentation process and updated during execution. To create and maintain this list, `Arancino` monitors Windows APIs related to virtual memory management, such as `NtAllocateVirtualMemory` and `NtWriteVirtualMemory`.

The `PinVMShield` extension provided in [15] has improved its original countermeasures by incorporating countermeasures against *Code Cache Fingerprints* and *Memory Region Permission Mismatches*. Both contermeasures are based on the monitoring of certain system functions related to memory inspection, as `Arancino` does.

Similarly, the tool introduced in [9] implements a memory access monitoring system that relies on lookup methods, outperforming the other tools mentioned. In addition, it also monitors Windows system calls contained in `win32k.sys`. As the allow-list, it uses a shadow page table as an array indexed by the page number. Allocation, release, memory permission change events, or any event that can cause address space changes results in a table update. When a violation is detected (that is, an application tries to interact with an address that does not belong to the allowed list), an exception is thrown.

Likewise, this tool also monitors assembly instructions related to indirect control-flow transfer (such as **`ret`** and **`jmp`** to registers or memory addresses) and checks at runtime if the destination addresses are in a memory region with execute permission. If so, the execution proceeds normally. Otherwise, an exception (protection page or memory access violation) is thrown. In practice, this verification is carried out at the same time as it is verified whether a memory address is present in the allowed list described above.

Countermeasure Against *IP in Unexpected Memory Registers.* One way to find out the address of the next assembly instruction to execute is to retrieve the value from the IP logical CPU register. This can be done using assembly instructions that save the execution context of the program, which includes the IP register in its content.

`PinVMShield` [61] and `Arancino` [57] have a countermeasure to monitor instructions that store the IP register value in memory, such as assembly instructions **`fstenv`** and **`fsave`**. In addition, `PinVMShield` monitors the **`fxsave`** instruction, while `Arancino` monitors the **`int`** `2e` instruction. When the execution of these instructions is detected, the value of the IP register stored in memory is patched with the expected value, so the program under analysis is deceived. Similarly, the tool introduced in [9] also monitors for assembly instructions as `PinVMShield` does that can leak the value of the IP register and replaces the code cache address with that of the original code. The monitoring capability of all three tools use the instruction granularity level provided by Pin.

Countermeasure Against *Incorrect Handling of SMC.* During the analysis of an application in Pin, the original application code is never executed, but rather a modified version that resides in code caches. Therefore, the DBI framework needs to handle the SMC appropriately to maintain the transparency of the DBI analysis.

In this regard, `Arancino` [57] detects changes made to instructions within the block in execution at runtime and then forces Pin to rebuild the code cache with the modified code. This detection is achieved by monitoring the write instructions that belong to a trace. For each instruction in a trace, the memory address that is written is checked. SMC is detected when the tool observes an attempt to write to the memory regions where the application code resides. Note that this analysis takes place at the time of the construction and execution of the trace, not at the time of the execution of each instruction.

At the time of writing, it is possible to enable or disable SMC support in all versions of Pin [20]. When enabled, Pin detects SMC and sends a callback when such an event occurs. Therefore, the analyst has the opportunity to activate code cache invalidation and then rerun the JIT compiler for the modified code. However, by default, Pin assumes that the basic blocks do not modify their own code, and therefore it does not check SMC at this level. To do this, the analyst must provide a special command line argument to force Pin to analyze SMC.

Countermeasure Against *Process Hierarchy.* This countermeasure consists of monitoring system calls that allow a program to obtain information about the process tree. On Windows, this can be done using `NtQuerySystemInformation` [40] or through the process `CSRSS.exe` [45].

Arancino intercepts calls to NtQuerySystemInformation and modifies their return values following a simple rule: the textual data containing the characters "pin.exe", if they exist, are replaced by "cmd.exe". Furthermore, the interactions between the application under analysis and the process CSRSS are denied by monitoring the NtOpenProcess API. A similar strategy is adopted in [10], in which APIs and system calls are monitored through hooks. In addition, they take additional measures to maintain consistency between sequences of invocations, grouping hooks into categories that associate the values to be masked with the replacement values.

Countermeasure Against *Fingerprints of DBI-related Binary Programs.* PinVMShield [61] detects and mitigates this type of evasion techniques by monitoring system calls and performing parameter checks. In particular, it uses a deny list made up of Pin-related strings. When one of these strings is detected as a parameter, a potential anti-instrumentation technique is identified and the analyst is responsible for overriding that parameter and continuing the call or otherwise dismissing the call.

Countermeasure Against *TLS Presence.* The extension provided in [15] adds to PinVMShield a countermeasure against *TLS Presence* that is based on the monitoring of system functions related to accessing TLS. However, advanced techniques to access TLS, such as specifically designed snippets of assembly code that access TLS through internal Windows structures, are not yet mitigated.

In particular,to mitigate this technique, the authors [15] take advantage of the fact the interactions with the TLS structure are done through OS APIs. Therefore, the approach is based on redirecting any attempt to read or write in positions used by the DBI framework to other positions. The authors built the countermeasure using the facilities provided by PinVMShield [61].

Countermeasure Against *System Library Hooks.* As mentioned before, DBI frameworks change some shared dynamic link libraries in the system by inserting hooks that divert the normal flow of execution to the instrumentation framework. The client application can easily detect these changes by checking the code in these system libraries. Additionally, these hooks can be avoided if the client application rewrites the code in these libraries appropriately.

Arancino [57] redirects to an unmodified version of these libraries when the client application queries on these libraries, ensuring the transparency of the hooks placed by the DBI framework. However, the execution of system calls within the libraries still targets the framework modified libraries.

Countermeasure Against *Common API Calls.* As discussed earlier, a client application can monitor the number of calls to APIs by adding multiple execution counters. To override detection techniques based on this approach, Arancino [57] monitors any read or write operation performed in the system routines commonly used by DBI frameworks. Therefore, any attempt to read or write in specific memory regions of the system libraries is redirected to another memory region to prevent the analyzed application from implementing monitoring artifacts in the system calls. The authors of Arancino claim that this approach is generic and can thwart all attempts to overwrite original function code or insert hooks into protected memory regions.

Countermeasure Against *Performance Degradation.* According to Polino et al. [57], mitigation of evasion techniques that use discrepancies in runtime measurements cannot depend solely on monitoring system functions that return system time (such as GetTickCount or timeGetTime, among others) since this information can also be recovered indirectly by other means. For example, a client application can query the shared memory page KUSER_SHARED_DATA, which provides resource usage and other information for the logged in user such as runtime [37]. In this regard, Arancino intercepts read attempts in the memory region where KUSER_SHARED_DATA resides and alters the return values to ensure that the existence of any temporary overhead is not revealed. The authors of Arancino also propose inserting a function hook into NtQueryPerformanceCounter, a system function that can be used to return elapsed time.

Another way to get an application runtime reference is by using the assembly instruction `rdtsc`, which returns the CPU processor timestamp in two general-purpose registers. As a countermeasure approach in this case, we can monitor at the instruction granularity level the execution of this instruction and change its return values by dividing them by a user-defined constant. Unfortunately, monitoring execution at such a low level would have a huge impact on analysis time.

Recently, D'Elia et al. have introduced [10], an extension of their earlier work in [9] where a fast forward time approach is proposed to intercept time stall operations (excluding those related to Windows libraries and internal functions) and the amount of time handled by these operations is stored in an internal data structure. Later, when the application queries the host time to determine the elapsed time after the operations, the result is appropriately modified to be consistent with the accumulated time in that internal data structure.

5 DISCUSSIONS

This section summarizes the discussion of our findings in the systematic review of the literature conducted. We first focus on evasion techniques and then discuss countermeasures.

5.1 On the Evasion Techniques

Table 2 summarizes the evasive techniques presented in this article and the papers that proposed them. In addition, it also shows if a PoC is available for each evasion technique.

A first finding is the existence of a small number of PoC. Among the 26 avoidance techniques presented, only 9 PoCs are provided in the literature. In the spirit of open science, only 4 papers have made PoC tools available to the public, namely [14] (dubbed `eXait`), [18], [25, 76] (dubbed `PwIN`), and [9].

Regarding the transparency property of DBI tools, all anti-instrumentation techniques look for artifacts in memory and in the system to detect the presence of a DBI framework. Therefore, DBI tools must be very careful and provide perfect transparency to go unnoticed. Regarding the isolation property of DBI tools, the anti-instrumentation techniques highlighted in Table 2 interact with resources strictly associated with DBI frameworks (such as code caches and TLS) as a form of detection. Therefore, DBI tools must be aware of this behavior and be prepared to counter it.

The current DBI frameworks' flaws in isolation and transparency properties when analyzing an application are the motivations for Kirsch et al. [25] and Zhechev [76] to consider them unsuitable for any security-related analysis. As shown, it is true that these properties are important for a DBI to remain stealthy. Note that some authors also pointed out that some evasive techniques may fail to detect DBI frameworks [61]. For instance, techniques associated with detection for runtime overhead, common API calls, or memory page permissions can lead to false positives. The possibility of these false positives can deter malware developers from using these evasive techniques.

Based on our findings, we argue that significant advances have been made to DBI framework protections to reduce the attack surface added during application analysis. These advancements make DBI suitable for certain types of security analysis (such as taint analysis, symbolic execution, or cryptoanalysis, to name a few), but make them unsuitable for others, such as sophisticated malware or advanced threats analysis. Currently, more efforts are needed in the design of DBI frameworks to achieve complete isolation and transparency when analyzing an application in the security context.

5.2 On the Countermeasures against Evasion Techniques

Table 3 lists the subset of anti-instrumentation techniques that have some countermeasure and in which articles and works they are proposed. Among the 26 techniques discussed in this work, only 12 countermeasures are proposed in the literature. Given the growing popularity of DBI frameworks, anti-instrumentation techniques

Table 2. Coverage of Anti-instrumentation Techniques by Taxonomies Proposed in the Literature Reviewed and in this Work

Evasive Techniques	**Articles and works**									**PoC**	**Classification**†
	[68]	**[30]**	**[18]**	**[14]**	**[61]**	**[57]**	**[25, 76]**	**[9]**	**(this work)**		
Unsupported Assembly Instructions	●								●	✗	FL
Unsupported Behaviors	●	●							●	✗	
Stalling Code					●				●	✗	RL
Memory Exhaustion									●	✗	
Code Cache Fingerprints	●					●	●	●	●	✓	CCAD
IP in Unexpected Memory Regions			●	●	●	●	●	●	●	✓	
Incorrect Handling of SMC			●			●	●	●	●	✓	
Unexpected Context	●								●	✗	
Memory Region Permission Mismatches							●	●	●	✓	
Process Hierarchy		●		●	●	●		●	●	✓	EAD
Xmode Code	●							●	●	✗	
Incorrect Emulation of Supported Assembly Instructions							●	●	●	✓	
Command-Line Arguments				●	●				●	✗	
Process Handles				●	●			●	●	✗	
File Handles		●						●	●	✗	
Event Handles				●	●			●	●	✗	
Shared Section Handles				●	●			●	●	✗	
Signal Masks		●							●	✗	
Fingerprints of DBI-related Binary Programs		●		●	●			●	●	✗	
TLS Presence	●							●	●	✗	
Environment Variables							●		●	✗	
System Library Hooks				●	●	●		●	●	✓	JCD
Excessive Number of Full Access Memory Pages	●			●	●	●	●		●	✓	
Common API Calls				●		●			●	✗	
Peak Memory Usage		●							●	✗	ROD
Performance Degradation		●		●	●	●	●	●	●	✓	

†FL: *Functional Limitation*; RL: *Resource Limitation*; CCAD: *Code Cache Artifact Detection*; EAD: *Environment Artifact Detection*; JCD: *JIT Compiler Detection*; ROD: *Runtime Overhead Detection.*

are expected to continue to attract the attention of both malware developers seeking counter-analysis solutions and researchers interested in finding appropriate countermeasures.

The mitigation techniques that are based on the monitoring of system calls have the disadvantage of the number of different calls to the system that provide the functionality to be monitored. This number of system

Table 3. Coverage of Countermeasures Against Anti-instrumentation Techniques by the Literature Reviewed

Evasive Techniques	**Articles and works**					**Classification**[†]
	[61]	[57]	[9]	[10]	[15]	
Code Cache Fingerprinting		●	●	●	●	
IP in Unexpected Memory Regions		●	●	●		
Incorrect Handling of SMC		●				CCAD
Unexpected Context		●				
Memory Region Permission Mismatches			●	●	●	
Process Hierarchy		●		●		
Fingerprints of DBI-related Binary Programs	●					EAD
TLS Presence					●	
System Library Hooks		●				
Excessive Number of Full Access Memory Pages		●	●			JCD
Common API Calls		●				
Performance Degradation	●			●		ROD

[†]CCAD: *Code Cache Artifact Detection*; EAD: *Environment Artifact Detection*; JCD: *JIT Compiler Detection*; ROD: *Runtime Overhead Detection.*

calls can be very large, depending on the specific OS and version. For example, Windows offers a more detailed interface for some library functions, which are usually postfixed with "Ex" (which means *extended*). Apart from these library functions, other Windows system calls contained in `win32k.sys` also need to be monitored, as the tool provided by D'Elia et al. [9] effectively does.

Additionally, some of the countermeasures presented do not cover all possible DBI evasion cases in all categories. For example, the solution proposed in [57] to mitigate runtime overhead does not contemplate the elapsed runtime calculated using external sources that depend on network connections, as indicated in [58, 61].

Another drawback is the overhead imposed by the countermeasures themselves on the application being analyzed: the lower the level of instrumentation granularity of the countermeasure, the greater the overhead. Although this additional overhead is a relevant metric in determining whether a countermeasure is usable in the real-world, not all the works have studied the overhead imposed by countermeasures. D'Elia et al. [9] and Santos Filho et al. in [15] perform the most detailed analysis, using the well-known SPEC CPU2006 benchmark, commonly used to evaluate the performance of DBI tools and more general tools [33]. Polino et al. [57] also conduct a performance analysis, but the authors do not detail the process. In contrast, neither Rodríguez et al. [61] nor D'Elia et al. [10] provide a performance analysis of their countermeasures. It should be noted that Santos Filho et al. in [15] evaluate the effectiveness of the new countermeasures added to `PinVMShield` (described in more detail in Section 4.3.2). However, we still lack a comprehensive study of the effectiveness and impact on the performance of other `PinVMShield` countermeasures. In this regard, it is worth mentioning the (more general) work in [60], where the performance of different DBI frameworks under different scenarios is evaluated. More research is needed to properly assess the performance of countermeasures and anticipate their applicability in real-world analysis scenarios.

Evasion techniques not mitigated at the time of writing include indirect evasion techniques (*Unsupported Assembly Instructions*, *Unsupported Behaviors*, *Stalling Code*, and *Memory Exhaustion*), all the direct evasion techniques based on environment artifacts (except the *Process Hierarchy*, *Fingerprints of DBI-related Binary Programs*, and *TLS Presence* evasion techniques), and half of the techniques based on runtime overhead detection (in particular, *Peak Memory Usage*).

As the rootkit paradox states [27], whenever code wants to run on a system, it must be visible to the system in some way. Therefore, all evasion techniques can be detected in some way. However, avoiding indirect evasion techniques can be difficult. For instance, mitigating *Unsupported Assembly Instruction* detection technique involves instrumenting at the instruction-granularity level. So we need to have this level of granularity supported by the DBI framework first. Additionally, fine-grained instrumentation such as instrumentation at the level of fetching assembly instructions has a large impact on performance, making it impractical for real-world scenarios. A similar issue problem to avoid techniques based on *Unsupported Behaviors*. Also, a detailed study is needed beforehand on what behaviors are not currently supported by DBI frameworks. As far as we know, this type of study has yet to be done. Likewise, a full study is needed to find the assembly instructions that are not currently supported, as well as the instructions that are supported but incorrectly emulated by DBI frameworks. In this regard, automatic and systematic techniques such as the approach proposed in [52] can be useful for this purpose. Regarding *Stalling Code* and *Memory Exhaustion*, we need to incorporate additional analysis abilities into current DBI frameworks (for instance, to analyze the semantics of code snippets within the client application).

With regard to avoiding *Xmode Code* and *Incorrect Emulation of Supported Assembly Instructions*, a similar problem arises: we need instruction-level instrumentation. For all other techniques for detecting environment artifacts, we can rely on specific system calls that allow an application to inspect command-line arguments or file handles, among others. A similar issue occurs with *Peak Memory Usage*. However, these solutions are incomplete as sophisticated evasion malware can also verify this information by walking through internal structures and thus be transparent to monitoring activity. Therefore, we again need instruction-level instrumentation with semantic analysis.

Finally, it is noteworthy that all the works that propose countermeasures [9, 10, 15, 57, 61] have made the source codes of their solutions available to the public so that they can be studied, used, and extended by other researchers and security professionals.

6 CHALLENGES AND OPEN ISSUES

Applications with malicious intent begin to embed code to detect analysis environments as a way to evade malware detection and classification. In this work, we have presented specific techniques designed for the detection of DBI frameworks, which are used for analysis of applications during their execution through instrumentation from a security perspective.

Despite the number of anti-instrumentation techniques (26 at the time of writing this article), a comprehensive review that compiled and classified all these techniques was lacking in the literature. In this article, we have attempted to close this gap by reviewing the literature to identify the existing set of DBI framework evasion techniques, as well as the existing set of countermeasures to avoid them. Furthermore, considering previous works in the literature, we have proposed a new taxonomy for anti-instrumentation techniques.

Our findings show that despite the significant advances made to DBI framework protections to reduce their attack surfaces, more efforts are still needed. Although DBI frameworks are currently suitable for certain security analyzes (such as taint analysis, symbolic execution, or cryptoanalysis, to name a few), at the same time they are unsuitable for others (such as the analysis of sophisticated malware or advanced threats). These efforts should focus on improving the design of the DBI frameworks to achieve complete isolation and full transparency, which are requirements needed when analyzing an application from a security perspective.

With the popularization of DBI frameworks for security analysis, the use of anti-instrumentation techniques is likely to grow, as has happened with other evasive techniques that detect virtual environments or debugging tools. We hope this work can help keep pace with the ongoing arms race between system advocates and malware developers by providing an overview of current anti-instrumentation techniques and proposed countermeasures to mitigate them.

During our research, we have found several gaps in the related literature that can be noted as challenges and interesting areas of future work:

(1) *Lack of countermeasures*: despite the best efforts of security tool developers, there are anti-instrumentation techniques that cannot be overcome and threaten the transparency of DBI tools, which represents a huge challenge for IT security professionals. In fact, less than half of the anti-instrumentation techniques reviewed in this work (12 out of 26) have any countermeasures. To ensure the protection of DBI frameworks

and the correctness of their analysis, it is desirable to have countermeasures for all anti-instrumentation techniques. In this regard, the survey in [5] indicates that future research on malware detection should focus on heuristics to detect evasive behavior. Additionally, existing countermeasures target specific DBI frameworks and OSs. Therefore, a generalization of these countermeasures is also needed.

(2) *Lack of experimentation in real-world scenarios*: as defined in [62], a real-world experiment is an evaluation scenario that incorporates the behavior of a significant number of *parts* that are in active use by persons other than the authors. While the countermeasures proposed in the literature are effective, as evidenced by PoC, they have been tested only on *synthetic* evasive malware (i.e., malware specially designed for this purpose). Therefore, there is a lack of assessment of the prevalence of these evasion techniques in actual malware samples. In this regard, the work that used a scenario most similar to a real one is [57], where countermeasures are used to detect evasive behavior in a set of $7,006$ real malware samples. However, as noted in [5], there are some difficulties that need to be carefully addressed with regard to acquiring real evasive malware. On the one hand, manual acquisition is inappropriate for large-scale classification of samples as evasive or non-evasive. On the other hand, automatic collection can be difficult as malware, by its very nature, can avoid detection. Similarly, autonomous acquisition systems like honeypots or upload-based sites (like VirusTotal) are unlikely to catch new malware specimens or very recent evasion techniques.

(3) *Lack of evaluation on the impact of countermeasures*: although security professionals and researchers make a great effort to develop countermeasures for anti-instrumentation techniques, the scientific work has little or no discussion about the overhead imposed by these countermeasures on DBI frameworks (especially with regard to effectiveness and performance). As noted in [62], analyzing the proportions of false positives and false negatives is interesting in revealing the strengths and weaknesses of any malware detection approach.

(4) *Lack of comparison between countermeasures*: as performance impacts and limitations of countermeasures are absent in many works, comparing these solutions becomes an arduous task. There is a gap in the literature regarding a detailed comparison between countermeasures, studying their effectiveness, and overheads in different scenarios. Such a comparison would provide a guide for developers to make it easier to adopt these solutions and choose one approach over another.

(5) *Lack of proofs of concept*: only 9 of the 26 anti-instrumentation techniques reviewed in this article have proofs of conceptPoC. This lack of PoC of anti-instrumentation techniques is a bad practice since they become essential for the validation and study of the effectiveness of these techniques and their corresponding countermeasures.

REFERENCES

[1] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *Proceedings of the Network and Distributed System Security Symposium.* The Internet Society, 16 pages.

[2] Derek Bruening. 2014. Translate the FPU instruction pointer Issue 698 DynamoRIO/dynamorio. Retrieved November 14, 2020 from https://github.com/dynamorio/dynamorio/issues/698.

[3] Derek Bruening and Saman Amarasinghe. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.D. Dissertation. Massachusetts Institute of Technology, Department of Electrical Engineering.

[4] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. 2001. Design and implementation of a dynamic optimization framework for windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization.* ACM, 12 pages.

[5] Alexei Bulazel and Bülent Yener. 2017. A survey on automated dynamic malware analysis evasion and counter-evasion: PC, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium.* ACM, New York, NY, Article 2, 21 pages.

[6] Cesar Cerrudo. 2005. Hacking windows internals. In *Proceedings of the Black Hat EU.*

[7] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing vulnerabilities. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security.* ACM, New York, NY, 334–345.

[8] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM, New York, NY, 40–51.

[9] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. ACM, New York, NY, 15–27.

[10] Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, and Lorenzo Cavallaro. 2020. On the dissection of evasive malware. *IEEE Transactions on Information Forensics and Security* 15 (2020), 2750–2765.

[11] Tevi Devor and Sion Berkowits. 2013. Pin: Intel's dynamic binary instrumentation engine. Retrieved November 14, 2020 from https://software.intel.com/content/dam/develop/external/us/en/documents/cgo2013-256675.pdf.

[12] Eldad Eilam and Elliot J. Chikofsky. 2005. *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons. 589 pages.

[13] Gustaf Ekenstein and David Norrestam. 2017. *Classifying Evasive Malware*. Master's thesis. Lund University.

[14] Francisco Falcón and Nahuel Riva. 2012. Dynamic Binary Instrumentation Frameworks: I know you're there spying on me. Retrieved November 14, 2020 from https://www.coresecurity.com/corelabs-research/open-source-tools/exait.

[15] Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. 2020. Reducing the attack surface of dynamic binary instrumentation frameworks. In *Proceedings of the Developments and Advances in Defense and Security*, Vol. 152. Springer, 3–13.

[16] Matthew Ryan Gilboy. 2016. *Fighting Evasive Malware With DVasion*. Master's thesis. University of Maryland.

[17] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, 571–585.

[18] Martin Hron and Jakub Jermář. 2014. SafeMachine malware needs love, too. Retrieved November 14, 2020 from https://www.virusbulletin.com/uploads/pdf/conference_slides/2014/sponsorAVAST-VB2014.pdf.

[19] Intel Corporation. 2011. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z. Retrieved November 14, 2020 from https://www.intel.es/content/www/es/es/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[20] Intel Corporation. 2019. Pin: Command Line Switches. Retrieved November 14, 2020 from https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html/group__KNOBS.html.

[21] Intel Corporation. 2021. Pin: Pin 3.18 User Guide - Callbacks. Retrieved November 14, 2020 from https://software.intel.com/sites/landingpage/pintool/docs/98332/Pin/html/index.html#CALLBACK.

[22] Intel Corporation. 2021. PinCRT overview PinCRT architecture. Retrieved from https://software.intel.com/sites/landingpage/pintool/docs/98332/PinCRT/PinCRT.pdf.

[23] Kaspersky Lab. 2017. Kaspersky Lab detects 360,000 new malicious files daily - up 11.5% from 2016. Retrieved December 18, 2018 from https://www.kaspersky.com/about/press-releases/2017_kaspersky-lab-detects-360000-new-malicious-files-daily.

[24] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, 287–301.

[25] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. 2018. PwIN – pwning intel piN: Why DBI is unsuitable for security applications. In *Proceedings of the 23rd European Symposium on Research in Computer Security*. Javier Lopez, Jianying Zhou, and Miguel Soriano (Eds.), Lecture Notes in Computer Science. Springer International Publishing, Cham, 363–382.

[26] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 285–296.

[27] Jesse D. Kornblum. 2006. Exploiting the rootkit paradox with windows memory analysis. *International Journal of Digital Evidence* 5, 1 (2006), 1–5.

[28] Yevgeny Kulakov. 2017. MazeWalker - Enriching Static Malware Analysis. Retrieved December 18, 2018 from https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-MazeWalker.pdf.

[29] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 412–425.

[30] Xiaoning Li and Kang Li. 2014. Defeating the transparency features of dynamic binary instrumentation. In *Proceedings of the BlackHat USA*.

[31] Linux Programmer's Manual. 2018. proc(5) - Linux manual page. Retrieved November 14, 2020 from http://man7.org/linux/man-pages/man5/proc.5.html.

[32] Gregory Lueck, Harish Patil, and Cristiano Pereira. 2012. PinADX: An interface for customizable debugging with dynamic instrumentation. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*. ACM, New York, NY, 114–123.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, New York, NY, 190–200.

[34] Pratyusa K. Manadhata. 2008. *An Attack Surface Metric*. Ph.D. Dissertation. Carnegie Mellon University.

[35] Sebastiano Mariani, Lorenzo Fontana, and Fabio Gritti. 2016. PinDemonium: A DBI-based generic unpacker for windows executables. In *Proceedings of the Black Hat USA*.

[36] McAfee Labs. 2019. McAfee Labs Threats Report, August 2019. Retrieved December 01, 2019 from https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf.

[37] Microsoft. 2017. kuser | Microsoft Docs. Retrieved November 14, 2020 from https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/-kuser.

[38] Microsoft. 2018. Event Objects (Windows). Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms682655(v=vs.85).aspx.

[39] Microsoft. 2018. File Handles (Windows). Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/aa364225(v=vs.85).aspx.

[40] Microsoft. 2018. NtQuerySystemInformation function (Windows). Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms724509(v=vs.85).aspx.

[41] Microsoft. 2018. Process Handles and Identifiers (Windows). Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms684868(v=vs.85).aspx.

[42] Microsoft. 2018. Running 32-bit Applications (Windows). Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/aa384249(v=vs.85).aspx.

[43] Microsoft. 2018. Thread Local Storage. Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms686749(v=vs.85).aspx.

[44] Microsoft. 2018. `main` function and command-line arguments. Retrieved November 14, 2020 from https://msdn.microsoft.com/en-us/library/88w63h9k.aspx.

[45] Microsoft Corporation. 2017. Microsoft Security Bulletin MS13-033 Microsoft Docs. Retrieved November 14, 2020 from https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2013/ms13-033.

[46] Microsoft Corporation. 2018. Creating Guard Pages - Win32 apps | Microsoft Docs. Retrieved November 14, 2020 from https://docs.microsoft.com/en-us/windows/win32/memory/creating-guard-pages.

[47] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 757–768.

[48] A. Moser, C. Kruegel, and E. Kirda. 2007. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference*. IEEE, 421–430.

[49] Nicholas Nethercote. 2004. *Dynamic Binary Analysis and Instrumentation*. Technical Report. University of Cambridge.

[50] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 44–66.

[51] NowSecure. 2015. FRIDA. Retrieved November 14, 2020 from https://frida.re/.

[52] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. USENIX Association, 2.

[53] Heidi Pan, Krste Asanović, Robert Cohn, and Chi-Keung Luk. 2005. Controlling program execution through binary instrumentation. *SIGARCH Computer Architecture News* 33, 5 (December 2005), 45–50.

[54] Paradyn. 2021. DynInst version 11.0.0. Retrieved November 14, 2020 from https://github.com/dyninst/dyninst.

[55] Mathias Payer. 2016. libdetox: A framework for online program transformation. In *Proceedings of the 2016 Workshop on Forming an Ecosystem Around Software Transformation*. 3 pages.

[56] Mathias Payer and Thomas R. Gross. 2013. Hot-patching a web server: A case study of ASAP code repair. In *Proceedings of the 2013 11th Annual Conference on Privacy, Security and Trust*. IEEE, 143–150.

[57] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and defeating anti-instrumentation-equipped malware. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment*. Michalis Polychronakis and Michael Meier (Eds.). Springer International Publishing, Cham, 73–96.

[58] Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. 2014. A framework for understanding dynamic anti-analysis defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*. ACM, New York, NY, Article 2, 9 pages.

[59] Quarkslab. 2018. QDBI. Retrieved November 14, 2020 from https://qbdi.quarkslab.com/.

[60] Ricardo J. Rodríguez, Juan Antonio Artal, and José Merseguer. 2014. Performance evaluation of dynamic binary instrumentation frameworks. *IEEE Latin America Transactions (Revista IEEE America Latina)* 12, 8 (December 2014), 1572–1580.

[61] Ricardo J. Rodríguez, Inaki Rodriguez Gaston, and Javier Alonso. 2016. Towards the detection of isolation-aware malware. *IEEE Latin America Transactions* 14, 2 (2016), 1024–1036.

[62] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. v. Steen. 2012. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. IEEE, 65–79.

[63] Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

[64] Alex Skaletsky, Tevi Devor, Nadav Chachmon, Robert Cohn, Kim Hazelwood, Vladimir Vladimirov, and Moshe Bach. 2010. Dynamic program analysis of microsoft windows applications. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software*. 2–12.

[65] Asia Slowinska, Istvan Haller, Andrei Bacs, Silviu Baranga, and Herbert Bos. 2014. Data structure archaeology: Scrape away the dirt and glue back the pieces!. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment*. Sven Dietrich (Ed.). Springer International Publishing, Cham, 1–20.

[66] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *Proceedings of the Revised Selected Papers of the 5th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes* . Vol. 8628, Springer-Verlag, Berlin, 155–167.

[67] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2015. Looking inside the black-box: Capturing data provenance using dynamic instrumentation. In *Proceedings of the Provenance and Annotation of Data and Processes*. Bertram Ludäscher and Beth Plale (Eds.). Springer International Publishing, Cham, 155–167.

[68] Ke Sun, Xiaoning Li, and Ya Ou. 2016. Break out of the truman show: Active detection and escape of dynamic binary instrumentation, 2016. In *Proceedings of the Black Hat Asia*.

[69] Mateus Tymburibá, Rubens E. A. Moreira, and Fernando Magno Quintão Pereira. 2016. Inference of peak density of indirect branches to detect ROP attacks. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, New York, NY, 150–159.

[70] Mateus Tymburibá, Ailtons S. Santos, and Eduardo L. Feitosa. 2014. Controlando a frequência de desvios indiretos para bloquear ataques ROP. In *Proceedings of the 2014 Brazilian Symposium on Information Security and Computational Systems*. SBC, Belo Horizonte, MG, Brasil, 223–236.

[71] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. 2019. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy Security* 22, 1 (Jan. 2019), Article 6, 30 pages.

[72] Vladimir Vladimirov. 2015. Re: Failure to instrument process tree. Retrieved November 14, 2020 from https://groups.io/g/pinheads/message/11807.

[73] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and emergency patch generation for integer overflow exploits. In *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment*. Sven Dietrich (Ed.). Springer International Publishing, Cham, 255–275.

[74] Zhilong Wang, Xuhua Ding, Chengbin Pang, Jian Guo, Jun Zhu, and Bing Mao. 2018. To detect stack buffer overflow with polymorphic canaries. In *Proceedings of the 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 243–254.

[75] Babak Yadegari and Saumya Debray. 2015. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, NY, 732–744.

[76] Zhechko Zhechev. 2018. *Security Evaluation of Dynamic Binary Instrumentation Engines*. Master's thesis. Department of Informatics, Technical Universtity of Munich.

Received November 2020; revised July 2021; accepted August 2021