SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed)

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Emilio Coppa
Sapienza University of Rome
coppa@diag.uniroma1.it

Simone Nicchi
Sapienza University of Rome
nicchi@diag.uniroma1.it

Federico Palmaro
Prisma
f.palmaro@prismaprogetti.it

Lorenzo Cavallaro
King's College London
lorenzo.cavallaro@kcl.ac.uk

ABSTRACT

Dynamic binary instrumentation (DBI) techniques allow for monitoring and possibly altering the execution of a running program up to the instruction level granularity. The ease of use and flexibility of DBI primitives has made them popular in a large body of research in different domains, including software security. Lately, the suitability of DBI for security has been questioned in light of transparency concerns from artifacts that popular frameworks introduce in the execution: while they do not perturb benign programs, a dedicated adversary may detect their presence and defeat the analysis.

The contributions we provide are two-fold. We first present the abstraction and inner workings of DBI frameworks, how DBI assisted prominent security research works, and alternative solutions. We then dive into the DBI evasion and escape problems, discussing attack surfaces, transparency concerns, and possible mitigations.

We make available to the community a library of detection patterns and stopgap measures that could be of interest to DBI users.

CCS CONCEPTS

• **Security and privacy → Systems security**; *Intrusion/anomaly detection and malware mitigation*; *Software reverse engineering*; *Software security engineering*.

KEYWORDS

Dynamic binary instrumentation, dynamic binary translation, interposition, transparent monitoring, evasion, escape

ACM Reference Format:
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 *ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3321705.3329819

AsiaCCS '19, July 9–12, 2019, Auckland, New Zealand
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6752-3/19/07…$15.00
https://doi.org/10.1145/3321705.3329819

1 INTRODUCTION

Even before the size and complexity of computer software reached the levels that recent years have witnessed, developing reliable and efficient ways to monitor the behavior of code under execution has been a major concern for the scientific community. Execution monitoring can serve a great deal of purposes: to name but a few, consider performance analysis and optimization, vulnerability identification, as well as the detection of suspicious actions, execution patterns, or data flows in a program.

To accomplish this task users can typically resort to instrumentation techniques, which in their simplest form consist in adding instructions to the code sections of the program under observation. One can think of at least two aspects that impact the instrumentation strategy that researchers can choose to support their analyses: the availability of the source code for the objects that undergo observation and the granularity of information that should be gathered. Additionally, a researcher may be interested in accessing instrumentation facilities that let them also alter the normal behavior of the program when specific conditions are observed at run time.

A popular instrumentation paradigm is represented by *dynamic binary instrumentation*. DBI techniques support the insertion of probes and user-supplied analysis routines in a running software for the sake of monitoring and possibly altering its execution up to the instruction level granularity, without requiring access to its source code or modifications to the runtime. The ease of use and flexibility that characterize DBI techniques has favored their adoption in an impressive deal of programming languages, software testing, and security research over the years.

Lately, the suitability of using DBI for security applications has been questioned in light of artifacts that popular DBI frameworks introduce in the execution, which may let an adversary detect their presence and cripple an analysis that hinges on them. This trend of research originated in non-academic forums like REcon and BlackHat where security researchers pointed out several attack surfaces for DBI detection and escape (e.g., [17, 24, 31, 56]).

Among academic works, Polino et al. [46] proposed countermeasures for anti-instrumentation strategies found in packers, some of which are specific to DBI. One year later Kirsch et al. [28] presented a research that instead deems DBI unsuitable for security applications, presenting a case study on its most popular framework.

Contributions. In this work we try to approach the problem of using DBI in software security research from a neutral stance, in hopes of providing our readers with insights on when the results of an analysis built on top of DBI should not be trusted blindly.

We distill the DBI abstraction, discuss inner workings and primitives of frameworks implementing it, and present a quantitative overview of recent security literature that uses DBI to back a heterogeneous plethora of analyses. We then tackle the DBI evasion and escape problems, discussing desired transparency properties and architectural implications to support them. We categorize known adversarial patterns against DBI engines by attack surfaces, and discuss possible mitigations both at framework design level and as part of user-encoded analyses. We collect instances of known adversarial sequences and prototype a mitigation scheme in the form of a high-level library that could be of interest to DBI users.

2 DYNAMIC BINARY INSTRUMENTATION

DBI systems have significantly evolved since the advent of DynInst [5] as a post-compilation library to support tools that wanted to instrument and modify programs during execution. In the following we present characteristic traits of the DBI abstraction and its embodiments, discussing popular frameworks and alternative technologies.

2.1 The DBI Abstraction

We can think of a DBI system as an application virtual machine that interprets the ISA of a specific platform (usually coinciding with the one where the system runs) while offering instrumentation capabilities to support monitoring and altering instructions and data from an analysis tool component written by the user:

Definition 2.1 (DBI System). A DBI system is an execution runtime running in user space for process-level virtualization. An unmodified compiled program executes within the runtime as would happen in a native execution, with introspection and execution control capabilities accessible to its users in a programmatic way.

The definition above is meant to capture distinctive traits of most DBI embodiments used in research in the last two decades. The components of a DBI runtime are laid out in the same address space where program execution will take place, with the program's semantics being carried out alongside user-supplied code for its analysis. Alternative designs recently proposed for moving the runtime outside the process where the code under analysis executes are discussed in Section 5.1 and Section 6.

Inner Workings of DBI Engines. A design goal for a DBI system is to make it possible to observe—and possibly alter—the entire architectural state of the program under analysis, including register values, memory addresses and contents, and control flow transfers.

To this end, the approach followed by most popular DBI embodiments is to recur to dynamic compilation: the original code of the application is not executed directly, but rather analyzed, instrumented and compiled using a just-in-time (JIT) compiler. An instruction fetcher component reads the original instructions in the program as they are executed for the first time, offering the engine the opportunity to instrument them before undergoing compilation.

The compilation unit is typically a *trace*, defined as a straight-line sequence of instructions ending in an unconditional transfer and possibly with multiple side exits representing conditional branches. Compiled traces are placed in a *code cache*, while a dispatcher component coordinates transfers among compiled traces and new fetches from the original code. Similarly as in tracing JIT compilers for language VMs, a trace exit can be linked directly to its target to bypass the dispatcher when a compiled version is available, while inline caching and code cloning strategies can be used to optimize indirect control transfers. Special care is taken for instructions that should not execute directly, such as those for system call invocations, as they get handled by an emulator component in a similar way to how privileged instructions are dealt with in virtual machine monitors for whole-system virtualization [20].

From the user's perspective, analysis code builds on instrumentation facilities exposed through an API interface, with the DBI backend taking care of program state switching between analysis routines and the code under observation.

The design space of a DBI engine accounts for different possibilities. An alternative to JIT compilation is the probe-based approach where the original program instructions are executed once they have been patched with trampolines to analysis code. In this work we deal with JIT-based engines only, as they can offer better performance for fine-grained instrumentation and are intuitively more transparent. Another choice is whether to operate on a native instruction set or by lifting code to an intermediate representation: the first choice typically can lead to faster compilation at the price of an increased complexity for the backend, while the other generally favors architectural portability.

Execution Correctness. One of the most critical challenges in the design of a DBI system is to prevent the native behavior of an application under analysis from inadvertently changing when executing inside the system. Real-world applications can exercise a good deal of introspective operations: common instances are retrieving instruction pointer values and return addresses for function invocations, and iterating over loaded code modules. When the execution environment orchestrated by the DBI runtime does not meet the expected characteristics, an application might exercise unexpected behaviors or most likely crash.

Bruening et al. [4] identify and discuss three broad categories of transparency requirements related to correctness: code, data, and concurrency. An example of code transparency when using JIT compilation is having every address manipulated by the application match the one expected in the original code: the DBI system translates addresses for instance when the OS provides the context for a signal or exception handler. Data transparency requires, e.g., exposing the CPU state to analysis code as it would be in a native execution (leaving the application's stack unhindered as the program may examine it) and not interfering with its heap usage. Concurrency transparency prescribes, e.g., that the runtime does not interfere by using additional locks or analysis threads.

Achieving these properties is difficult when handling generic code, as a system should not make assumptions on how a program has been compiled or how it manipulates registers, heap and stack: for instance, some versions of Microsoft Office read data or execute code located beyond the top of the stack, while aggressive loop optimizations may use the stack pointer to hold data [4].

Primitives for Analysis. DBI systems offer general-purpose APIs that can accommodate a wide range of program analyses, allowing users to write clients (most commonly referred to as *DBI tools*) that run interleaved with the code under analysis. One of the reasons behind the vast popularity of DBI frameworks is that DBI architects tried not to put too many restrictions on tool writers [4]: although sometimes users may miss the most effective way for the job, they are not required to be DBI experts to implement their program analyses. The runtime exposes APIs to observe the architectural state of a process (e.g., memory and register contents, control transfers) from code written in traditional programming languages such as C/C++, often supporting the invocation of external libraries (e.g., for disassembly or constraint solving). A DBI system may also try to abstract away idiosyncrasies of the underlying instruction set, providing functions to intercept generic classes of operations such as reading from memory or transferring control flow.

From the client perspective, a generic DBI system provides primitives to handle at least the following elements and events:

- instructions in the original program to be instrumented;
- system call invocation, before and after a context switch;
- library function invocation, intercepted at the call site and also on return when possible (think of, e.g., tail calls);
- creation and termination of threads and child processes;
- dynamic code loading and unloading;
- exceptional control flow;
- asynchronous control flow (e.g., callbacks, Windows APCs).

Instructions can be exposed to the client when traces are built in the code cache, allowing it to iterate over the basic blocks that compose them, or when code images are first loaded to memory, enabling ahead-of-time (AOT) instrumentation. AOT instrumentation is useful for instance when analyzing libraries (e.g., to place hooks at the beginning of some functions), but cannot access information like basic-block boundaries that is revealed only at run time.

The capabilities offered to a client are not limited to execution inspection, but include the possibility of altering the program behavior. Common examples are overwriting register contents, replacing instructions, and modifying the arguments and return values for a function call or rewiring it to some user-defined function.

Sophisticated engines like Pin and DynamoRIO assist users in tool creation by providing facilities for memory manipulation, thread local storage, creation of analysis threads, synchronization, and interaction with the OS (e.g., for file creation) that minimize the possibility of interfering with the execution of the application.

The DBI abstraction can cope with sequences of code interleaved with data, overlapping instructions, statically unknown targets for indirect branches, and JIT code generation on the application side. Another appealing feature is the possibility for some engines to attach to a process and then release it just like a debugger; this might come in handy, e.g., for large, long-running applications [34].

2.2 Popular DBI Frameworks

Pin [34], DynamoRIO [4] and Valgrind [39] are possibly the most popularly known DBI frameworks, supporting different architectures and operating systems. They have been extensively used in countless academic and industrial projects, providing reliable foundations for building performant and accurate analysis tools.

Pin provides robust support for instrumenting binary code running on Intel architectures. Its instrumentation APIs allow analysis tools to register for specific statements (e.g., branch instructions) or events (e.g., thread creation) callbacks to analysis routines that can observe the architectural state of the program. A recurrent criticism is related to its closed source nature as it limits possible extensions.

DynamoRIO is an open source project and unlike Pin it exposes the entire instruction stream to an analysis tool, allowing users to perform many low-level code transformations directly. The superior performance level it can offer compared to Pin is still a popular subject of discussion within the DBI community.

Sometimes the analysis code might be coupled too tightly with details of the low-level binary representation. Valgrind and DynInst approach this problem in different ways. Valgrind lifts binary code to an architecture-independent intermediate representation: its developers could port it to many platforms to the benefit of analysis tools based on it. However, this comes with performance penalties that could make it inadequate in several application scenarios. DynInst tries to provide high-level representations of the analyzed program to the analysis tool: by exposing familiar abstractions such as the control-flow graph, functions, and loops, DynInst makes it easier to implement even complex analyses. However, the intrinsic difficulties in the static analysis work required to back them may lead DynInst to generate incomplete representations in presence of, e.g., indirect jumps or obfuscated code sequences.

Frida [27] tries to ease DBI tool writing by letting users write analysis code in JavaScript, executing it within the native application by injecting an engine. The framework targets quick development of analysis code, aiming in particular at supporting reverse engineering tasks. Due to its flexible design, Frida can support several platforms and architectures, including for instance mobile ones, but its intrusive footprint could be a source of concern.

While for most DBI frameworks guest and host architectures coincide, Strata [51] uses software dynamic translation to support different host and guest ISAs, requiring users to implement only few guest and host-specific components. Valgrind could technically support different ISAs, but its current implementation does not.

libdetox [40] featured the first DBI framework design concerned with transparency for security uses. Originally used as a foundation for a user-space sandbox for software-based fault isolation, libdetox randomizes the location of the code cache and other internal structures, posing particular attention on, e.g., preventing internal pointers overwriting and disabling write accesses to the code cache when the program executes. At least for its publicly released codebase, the framework is however vulnerable to some of the attacks that we will describe throughout Section 4.

Although more DBI frameworks [37, 48, 49] appeared recently, their designs did not introduce architectural changes relevant for security uses: for this reason we opted for not covering them.

2.3 Alternative Technologies

Source code instrumentation. When the source code of a program is available and the analysis is meaningful regardless of its compiled form[1], instrumentation can take place at compilation time. Analyses can thus be encoded using source-to-source transformation languages like CIL [38] or by resorting to compiler assistance. The most common instance of the latter approach are performance profilers, but in recent years a good deal of sanitizers have been built for instance on top of the LLVM compiler.

Operating at source level offers a few notable benefits. Analysis code can be written in an architecture-independent manner, also

[1] There are cases where the source may not be informative enough (e.g., for side channel detection) or Heisenberg effects appear if the source is altered (e.g., for memory errors).

allowing it to access high-level properties of an application (e.g., types) that could be lost during compilation. Also, instrumentation code undergoes compiler optimization, often leading to smaller performance overheads. However, when the scope of the analysis involves interactions with the OS or other software components the applicability of source code instrumentation may be affected.

Static binary instrumentation. A different avenue to instrument a program could be rewriting its compiled form statically. This approach is commonly known as Static Binary Instrumentation (SBI), and sometimes referred to as binary rewriting. While techniques for specific tasks such as collecting instruction traces had already been described three decades ago [16], it is only with the ATOM [55] framework that a general SBI-based approach to tool building was proposed. ATOM provided selective instrumentation capabilities, with information being passed from the application program to analysis routines via procedure calls. SBI generally provides better performance than DBI, but struggles in the presence of indirect branches, anti-disassembly sequences, dynamically generated code (JIT compiled or self-modifying), and shared libraries. ATOM was shortly followed by other systems (e.g., [29, 50, 67]) that gained popularity in the programming languages community especially for performance profiling tasks before the advent of DBI. Recent research [66] has shown how to achieve with SBI some of the practical benefits of DBI, such as instrumentation completeness along the software stack and non-bypassable instrumentation. Some obfuscation techniques and self-modifying code remain however problematic, causing execution to terminate when detected.

Cooperation on the runtime side. Another possibility is to move the analysis phase on the runtime side, like the virtual machine for managed languages or the operating system for executables. While the former possibility has been explored especially in programming languages research (e.g., using instrumentation facilities of Java VMs), the latter has seen several uses in security, for example in malware analysis to monitor API calls using a hooking component in kernel space. Although finer-grained analyses like instruction recording or information flow analysis are still possible with this approach [15], the flexibility of an analysis component executing in kernel mode is more limited compared to DBI and SBI.

Virtual machine introspection. In recent years a great deal of research has adopted Virtual Machine Introspection (VMI) techniques to perform dynamic analysis from outside the virtualized full software stack in which the code under analysis runs. The VMI approach has been proposed by Garfinkel and Rosenblum [20] to build intrusion detection systems that retain the visibility of host-based solutions while approaching the degree of isolation of network-based ones, and became very popular ever since. Inspecting a virtual machine from the outside enables scenarios such as code analysis in kernel space that are currently out of reach of DBI systems (an attempt is made in [65]). VMI is possible for both emulation-based and hardware-assisted virtualization solutions, allowing for different trade-offs in terms of execution speed and flexibility of the analysis. Unlike DBI, VMI incurs a *semantic gap* when trying to inspect high-level concepts of the guest system such as API calls or threads. Recent research has thus explored ways to minimize the effort required to build VMI tools, e.g., with automatic techniques [14] or by borrowing components from memory forensic frameworks [45]. While the ease of use of VMI techniques has lately improved with the availability of scriptable execution frameworks [57], performing analyses that require deep inspection features or altering properties of the execution other than the outcome of CPU instructions (say replacing a function call) remains hard for a user, or at least arguably harder than in a DBI system. We will return to this matter in Section 6 discussing also transparency.

3 DBI IN SECURITY RESEARCH

To provide the reader with a tangible perspective on the ubiquity of DBI techniques in security research over the years, we have reviewed the proceedings of flagship conferences and other popular venues looking for works that made use of them. Although this list may not be exhaustive, and a meticulous survey of the literature could be addressed in a separate work, we identified 95 papers and articles from the following venues: 18 for CCS, 7 for NDSS, 6 for S&P, 14 for USENIX Security, 10 for ACSAC, 4 for RAID, 4 for ASIA CCS, 2 for CODASPY, 9 for DIMVA, 7 for DSN, 3 for (S)PPREW, and 11 among ESSoS, EuroSys, ICISS, ISSTA, MICRO, STC, and WOOT.

Prominent applications. For the sake of presentation, we classify these works in the following broad categories, reporting the most common types of analysis for each of them:

- *cryptoanalysis*: identification of crypto functions and keys in obfuscated code [30], obsolete functions replacement [2];
- *malicious software analysis*: e.g., malware detection and classification [36, 64], analysis of adversarial behavior [43], automatic unpacking [46];
- *vulnerability analysis*: e.g., memory errors and bugs [9, 59], side channels [63], fuzzing [8], prioritization of code regions for manual inspection [22], debugging [61];
- *software plagiarism*: detection of unique behaviors [60];
- *reverse engineering*: e.g., code deobfuscation [53], protocol analysis and inference [6, 32], configuration retrieval [58];
- *information flow tracking*: design of taint analysis engines and their optimization [25, 26];
- *software protection:* e.g., control flow integrity [47], detection of ROP sequences [12], software-based fault isolation [41], code randomization [23], application auditing [68].

This choice left out 3 works that dealt with protocol replay, code reuse paradigms, and hardware errors simulation, respectively.

Categories can have different prevalence in general conferences: for instance, 6 out of 14 USENIX Security papers deal with vulnerability detection, but only 2 of the 18 CCS papers fall into it. For a specific category, works are quite evenly distributed among venues: for instance, works in malicious software analysis (14) have appeared in CCS (5), DIMVA (4), and five other conferences.

The heterogeneity of analyses built on top of DBI engines is somehow indicative of the flexibility provided by the DBI abstraction to researchers for prototyping their analyses and systems. The first works we surveyed date back to 2007, with 7 papers in that year. The numbers for the past two years (9 works in 2017, 8 in 2018) are lined up with those from four years before that (9 in 2013, 8 for 2014), hinting that DBI is still very popular in security research.

Usage of DBI primitives. For each work we then identify which DBI system is used and what primitives are necessary to support the

APPLICATION DOMAIN	DBI PRIMITIVES									WORKS
	INSTRUCTIONS				SYSTEM CALLS	LIBRARY CALLS	THREADS & PROCESSES	CODE LOADING	EXCEPTIONS & SIGNALS	
	MEMORY R/W	CALLS/RETS	BRANCHES	OTHER						
Cryptoanalysis	✓	✓	✓	✓						7
Malicious Software Analysis	✓	✓	✓	✓	✓	✓	✓	✓	✓	14
Vulnerability Detection	✓	✓	✓	✓	✓	✓				22
Software Plagiarism	✓				✓					2
Reverse Engineering	✓	✓	✓	✓	✓	✓				9
Information Flow Tracking	✓	✓	✓	✓		✓			✓	8
Software Protection	✓	✓	✓	✓	✓	✓	✓	✓	✓	30

Table 1: Application domains and uses of DBI primitives in related literature.

proposed analysis. We grouped instrumentation actions required by analyses in the following types:

- *instructions*, which we further divide in memory operations (for checking every register or memory operand’s content), call/ret (for monitoring classic calls to internal functions or library code), branches ([in]direct and [un]conditional), and other (for special instructions such as `int` and `rdtsc`);
- *system calls*, to detect low-level interactions with the OS;
- *library calls*, when high-level function call interception features of the DBI engine (like the routine instrumentation of Pin) are used to identify function calls to known APIs;
- *threads and process*, when the analysis is concerned with intercepting their creation and termination;
- *code loading*, to intercept dynamic code loading events;
- *exceptions and signals*, to deal with asynchronous flows.

In Table 1 we present aggregate results for application domains, where for each category we consider the union of the instrumentation primitives used by the works falling into it. While such information is clearly coarse-grained compared to a thorough analysis of each work, we observed important regularities within each category. For instance, in the cryptoanalysis domain nearly every considered work resorts to all the primitives listed for the group. Tracing all kinds of instructions and operands seems fundamental in the analysis of malicious software, while depending on the goal of the specific technique it may be necessary to trace also context switches or asynchronous flows. Observe that some primitives are intuitively essential in some domains: this is the case of memory accesses for information flow analysis, as well as of control transfer instructions in software protection works.

Choices. We identify Pin as the most popular engine in the works we survey with 57 uses, followed by Valgrind (19), libdetox (5), DynamoRIO (4) and Strata (3); in some cases the engine was not reported. Unsurprisingly, Valgrind is very popular in the vulnerability detection domain with 9 uses, just behind Pin with 12.

An important design choice that emerged from many works is related to when DBI should be used to back an online analysis or to rather record relevant events and proceed with offline processing. Other than obvious aspects related to the timeliness of the obtained results (e.g., shepherding control flows vs. bug identification) and nondeterministic factors in the execution, also the complexity of the analysis carried out on top of the retrieved information may play a role—this seems at least the case with symbolic execution.

A few research works [44, 68] aiming at mitigating defects in software devise their techniques in two variants: one for when the source code is available, and another based on DBI for software in binary form, hinting at higher implementation effort and possible technical issues in using SBI techniques in lieu of DBI.

4 ATTACK SURFACES

In light of the heterogenous analyses mentioned in the previous section, it is legitimate to ask whether their results may be affected by imperfections or lack of transparency of the underlying DBI engine. We will deal with these issues throughout the present section.

4.1 Desired Transparency Properties

Existing literature has discussed the transparency of runtime systems for program analysis under two connotations. The first, which is compelling especially for VM architects and dynamic translator builders, implies that an application executes as it would in a non-instrumented, native execution [4], and that interoperability with native applications works normally [11]. To this end, Bruening et al. [4] identify three guidelines to achieve the execution correctness properties outlined in Section 2.1 when writing a DBI system:

[G1] leave the code unchanged whenever possible;
[G2] when change is unavoidable, ensure it is imperceivable to the application;
[G3] avoid resource usage conflicts.

The authors explain how transparency has been addressed on an ad-hoc basis in the history of DBI systems, as applications were found to misbehave due to exotic implementation characteristics with respect to code, data, concurrency, or OS interactions.

DBI architects are aware that absolute transparency may be unfeasible to obtain for certain aspects of the execution, or that implementing a general solution would cause a prohibitive overhead. In the words of Bruening et al. [4]: *“the further we push transparency, the more difficult it is to implement, while at the same time fewer applications require it”*. In the end, the question boils down to seeing whether a presumably rare corner case may show up in code analyzed by the users of the DBI system.

A second connotation of transparency, which is compelling for software security research, implies the possibility of adversarial sequences that look for the presence of a DBI system and thwart any analyses built on top of it. We defer the discussion of the DBI evasion and escape problems to the next section, as in the following we will revisit from the DBI perspective general transparency properties that authors of seminal works sought in analysis runtimes they considered for implementing their approaches.

For an IDS, Garfinkel and Rosenblum [20] identify three capabilities required to support good visibility into a monitored system while providing significant resistance to both evasion and attack:

- *isolation*: code running in the system cannot access or modify code and data of the monitoring system;
- *inspection*: the monitoring system has access to all the state of the analyzed system;

- *interposition*: the monitoring system should be able to interpose on some operations, like uses of privileged instructions.

The last two properties are simply met by the design goals behind the DBI abstraction. Supporting the first property in the presence of a dedicated adversary seems instead hard for a DBI engine: as the runtime shares the same address space of the code under analysis, the possibility of covert channels is real. Actually, by subverting isolation an attacker might in turn also foil the inspection and interposition capabilities of a DBI system [28].

The authors of the Ether malware analyzer [13] present a simple abstract model of program execution where transparency is achieved if the same trace of instructions is observed in an environment with and without an analyzer component present. As they use hardware virtualization, the model is later generalized to account for virtual memory, exception handling, and instruction privilege levels. The requirements identified for a system that wants to hide memory and CPU changes caused by its own presence are:

- *higher privilege*: the analyzer runs at a privilege level higher than the highest level a program can achieve;
- *privileged access for side effects*: if any, side effects can be seen only at a privilege level that the program cannot achieve;
- *same basic instruction semantics*: aside from exceptions, the semantics of an instruction is not involved in side effects;
- *transparent exception handling*: when one occurs, the analyzer can reconstruct the expected context where needed;
- *identical timing information*: access to time sources is trapped, so that query results can be massaged consistently.

For an analysis runtime operating in user space, fulfilling all these requirements simultaneously is problematic, and oftentimes unfeasible. While DBI systems preserve instruction semantics and can capture exceptions, current embodiments operate at the same privilege level of the code under analysis. Values retrieved from timing sources can be massaged as in [46] to deal with specific detection patterns, but general strategies for manipulating the time behavior of a process with realistic answers may be intrinsically difficult to conceive or computationally too expensive [19].

For a fair comparison, we observe that similar problems affect to some extent also other other analysis approaches whose design seems capable of accommodating such requirements in a robust manner. Let us consider VMI techniques: Garfinkel in [19] describes several structural weaknesses in virtualization technology that an attacker may leverage to detect its presence, concluding that building a transparent monitor *"is fundamentally infeasible, as well as impractical from a performance and engineering standpoint"*. Attacks to VMI-based analyses are today realistic: for instance, performance differences can be observed due to TLB entry eviction [3], and the falsification of timing information can be imperfect [42].

In the next sections we will present the reader with practical attack surfaces that a dedicated adversary may use to detect a DBI system, and discuss how analyses running on one can be impacted.

4.2 DBI Evasion and Escape

The imperfect transparency of DBI systems has led researchers to design a plethora of detection mechanisms to reveal the presence of a DBI framework from code that undergoes dynamic analysis. Once an adversary succeeds, the code can either execute misleading actions to deceive the analysis, or attempt to carry out execution flows that go unnoticed by the engine. These scenarios are popularly known as the DBI *evasion* and *escape* problem, respectively.

Definition 4.1 (DBI Evasion). A code is said to evade DBI-based analysis when its instruction and data flows eventually start diverging from those that would accompany it in a native execution, either as a result of a decision sequence that revealed the presence of a DBI engine, or because of translation defects on the DBI side.

We opted for a broad definition of the evasion problem for the following reason: alongside techniques that actively fingerprint known artifacts of a DBI engine and deviate the standard control flow accordingly, DBI systems suffer from translation defects that are common in binary translation solutions and cause the analysis to follow unfeasible execution paths. The most prominent example is the use of self-modifying code, which is used both in benign mainstream programs [4] (resulting in an unintended evasion, and possibly in a program crash) and as part of implicit evasion mechanisms to cripple dynamic analysis by yielding bogus control flows.

Definition 4.2 (DBI Escape). A code is said to escape DBI-based analysis when parts of its instruction and data flows get out of the DBI engine's control and execute natively on the CPU, in the address space of the analysis process or of a different one.

An attacker aware of the presence of a DBI engine may try to hijack the control transfers that take place under the DBI hood, triggering flows that may never return under its control. The typical scenario involves leaking an address inside the code cache and patching it with a hijacking sequence, but more complex schemes have been proposed. As for the second part of the definition, DBI frameworks can provide special primitives to follow control flows carried out in other processes on behalf of the code under analysis: for instance, Pin can handle child processes, injected remote threads, and calls to external programs. Implementation gaps are often the main reason for which such attempts could go unnoticed.

4.3 Artifacts in Current DBI Embodiments

In Section 1 we have mentioned several scientific presentations and academic research highlighting flaws in DBI systems. While some of them characterize the DBI approach in its generality, others leverage implementation details of a specific engine, but can often be adapted to others that follow similar implementation strategies.

In hopes of providing a useful overview of the evasion problem to researchers that wish to use DBI techniques in their works, we propose a categorization of the techniques that are known to date to detect the presence of a DBI system. We will refer to Pin on Windows in many practical examples, as the two have received significantly more attention than any other engine/OS in the DBI evasion literature. We will discuss the following attack surfaces that we identified in such research: *time overhead*, *leaked code pointers*, *memory contents and permissions*, *DBI engine internals*, *interactions with the OS*, *exception handling*, and *translation defects*.

Time overhead. The process of translating and instrumenting the original instructions in traces to be placed in cache and eventually executed introduces an inevitable slowdown in the execution. This slowdown grows with the granularity of the required analysis: for example, tracing memory accesses is significantly more expensive than monitoring function calls. An adversary may try generic time measurement strategies for dynamic analyses that compare the execution of a code fragment to one in a reference system and look for significant discrepancies. There are peculiarities of DBI that could be exploited as well: for instance, the time required to dynamically load a library from user code can be two orders of magnitude larger [17] under DBI due to the image loading process. Similarly, effects of the trace compilation process can be exploited by observing fluctuations in the time required to take a branch in the program from the first time it is observed in the execution [28].

Leaked code pointers. A pivotal element in the execution transparency of DBI is the decoupling between the real instruction pointer and the virtual one exposed to the code. There are however subtle ways for an adversary to leak the real IP and compare it against an expected value. One way inspired by shellcode writing is to use special x87 instructions that are used to save the FPU state (e.g., for exception handling) in programs: an adversary executes some x87 instruction (like pushing a number to the FPU stack) and then uses `fstenv`, `fsave`, or one of their variants to write the FPU state to memory. The materialized structure will contain the EIP value for the last performed FPU instruction, which DBI engines typically do not mask: a check on its range will thus expose DBI [17]. Another way in 32-bit Windows is the `int 2e` instruction normally used to enter kernel mode on such systems: by clearing EAX and EDX before invoking it, the real IP is leaked to EDX [46].

Memory contents and permissions. A major source of transparency concerns is that a DBI engine shares the same address space of the analyzed code without provisions for isolation. The inspection of the address space can reveal additional sections and unexpected exported functions from the runtime [17]; the increased memory usage could be an indicator of the presence of DBI as well [31]. An adversary may look for recurrent patterns that are present in the code components (the runtime and the user's analysis code) of the DBI system and in their data, or in heap regions used for the code caching mechanism [17]. Another issue could be the duplicate presence of command-line arguments in memory [17].

Also the memory layout orchestrated and exposed by the DBI engine to the application under analysis can be stressed for consistency by adversarial sequences. For instance, Pin and DynamoRIO miss permission violations when the virtual IP falls into code pages for which access has been disabled (PAGE_NOACCESS) or guarded (PAGE_GUARD) by the application [24]. Similarly, an engine may erroneously process and execute code from pages that were not granted execution permissions [28].

DBI engine internals. While the CPU context exposed to the application under analysis is masked by the DBI abstraction, some changes applied to the execution context to assist a DBI runtime are not. One instance is represented by Thread Local Storage slots: developers can use TLS to maintain thread-specific storage in concurrent applications, but for the sake of efficiency DBI engines may occupy slots with internal data [56]. Another attack surface is represented by DLL functions that are altered by the engine: while the vast majority of library code goes unhindered through the DBI engine, in special cases trampolines should be inserted at their head. In the case of Pin on Windows systems `ntdll` is patched for [52]:

- `KiUserExceptionDispatcher`, to distinguish exceptions in the running code from internal (engine/analysis code) ones;
- `KiUserApcDispatcher` and `KiUserCallbackDispatcher`, as the Windows kernel can deliver asynchronous events;
- `LdrInitializeThunk`, to intercept user thread creation.

Interactions with the OS. DBI engines are concerned with the transparency of the execution space of an application, but as they are userspace VMs their presence can be revealed by interacting with the OS. A classic example is to check for the parent process [17, 31] or the list of active processes to reveal Pin or DynamoRIO. Handles can reveal the presence of a DBI engine too, for instance when fewer than expected are available for file manipulation [31] or when their names give away the presence of, e.g., Pin [17]. We found out that Pin may be revealed also by anti-debugging techniques based on `NtQueryInformationProcess` and `NtQueryObject`.

Exception handling. DBI engines have capabilities for hooking exceptional control flow: for instance, this is required to provide SEH handlers with the same information that would accompany the exception in a native execution [52]. There are however cases that DBI embodiments may not deal with correctly. For instance, we found out that Pin may not handle properly single-step exceptions and `int 2d` instructions used in evasive malware, with the sample not seeing the expected exception.

Translation defects. Analysis systems that base their working on binary translation are subject to implementation defects and limitations: this is true for DBI but also for full system emulators like QEMU. A popular example is the `enter` instruction that is not implemented in Valgrind [28]. DBI architects may decide to not support rarely used instructions; however, some instructions are intrinsically challenging for DBI systems: consider for instance far returns, which in Pin are allowed only when within the same code segment. Similarly, Pin cannot run "Heaven's gate code" to jump into a 64-bit segment from a 32-bit program by altering the CS selector. DynamoRIO does not detect the change, paving the way to the Xmode code evasion [56] that uses special instructions having the same encoding and disassembly on both architectures to yield different results due to the different stack operations size.

We put in this category also uses of self-modifying code (SMC): intuitively, SMC should always lead to invalidation of cached translated code, but implementations may miss its presence. In 2010 Martignoni et al. reported: *"the presence of aggressive self-modifying code prevents [...] from using efficient code emulations techniques, such as dynamic binary translation and software-based virtualization"* [35]. Detecting SMC sequences affecting basic blocks other than the current is nowadays supported by many DBI engines, while SMC on the executing block complicates the picture: DynamoRIO handles it correctly, while Pin has caught up from its 3.0 release providing a strict SMC policy option to deal with such cases.

4.4 Escaping from Current DBI Embodiments

To the best of our knowledge, the first DBI escape attack has been described in a 2014 blog post by Cosmin Gorgovan [21]: the author investigated how weaknesses of current DBI implementations highlighted in evasion-related works could pave the way to DBI escape. Intuitively, an avenue is represented by having code cache locations readable and writable also by the code under analysis.

Instead of leveraging a leaked pointer artifact, the author suggests encoding a fairly unique pattern in a block that gets executed and looking it up in the code cache once the latter's randomized location is determined by querying the OS memory map. The pattern gets overwritten with a trampoline, and when execution reaches the block again the code escapes. Gorgovan also explains how to make execution gracefully return under the control of the DBI system.

The code cache is not the only region that can be tampered with: escape attacks involving internal data, callbacks, and stack that are specific to a DBI engine are described in [56]. More recently, [28] describes a complex data-only attack for COTS applications running in Pin that uses a known memory corruption bug to escape, leveraging relative distances between regions that host, e.g., the libc and the code cache that are predictable in some Linux releases.

5 COUNTERMEASURES

In the following we investigate how transparency problems of DBI can be mitigated by revisiting the design space of engines, and when stopgap measures shipped with user code could be helpful.

5.1 Design Space of DBI Frameworks

The design of general-purpose embodiments of the DBI abstraction has historically been driven by the necessity of preserving execution correctness while obtaining an acceptable performance level. While general techniques for dynamic code generation and compilation have dramatically improved in the past two decades in response to the ever-growing popularity of languages for managed runtimes, DBI architects have to face additional, compelling execution transparency problems. They thus strove to improve the designs and implementations of their systems as misbehaviors were observed in the analysis of mainstream applications [4], backing popular program analysis tools such as profilers, cache simulators, and memory checkers.

The security research domain is however characterized by application scenarios where the program under inspection may resort to a plethora of generic or DBI-specific techniques to elude the analysis or even tamper with the runtime. A researcher may thus wonder how the design of a general-purpose DBI framework could be adapted to deal with common categories of adversarial sequences. In some cases the required changes could not be easy to be accommodated by a DBI framework with a wide audience of users from different domains, but could be sustainable for an engine that is designed with specific security applications in mind: one prominent example is SafeMachine [24], a proprietary DBI framework used by the Avast security firm for fine-grained malware analysis.

We will now revisit the attack surfaces from Section 4.3 from a DBI architect's perspective, referencing research works that have dealt with specific aspects and discussing other viable options.

Time overhead. Dealing with the run-time overhead from a dynamic analysis is an old problem in research. The overhead of a DBI system is not easy to hide, and it may not only be revealed by adversarial measurement sequences, but as discussed in [4] can also affect the correct execution of code sensitive to time changes. Also, the time spent in analysis code might exceed the cost of mere instrumentation depending on the type of analysis carried out.

Previous research in malware analysis has explored mechanisms to alter the time behavior perceived by the process by faking the results of time queries from different sources [46]. However, their efficacy is mostly tied to detection patterns observed in a specific domain. A general solution based on realistic simulations of the time elapsed in executing instructions as if the cost of DBI were evicted is believed impractical for dynamic analyses [19]. Also, such schemes may be defeated by queries to other processes not running under DBI or to external attacker-controlled time sources. Compared to VMI solutions where one may patch the time sources of the entire guest machine, DBI architects are thus left with (possibly much) less wiggle room to face timing attacks.

Leaked code pointers. We have mentioned subtle ways to leak code cache addresses through execution artifacts for specific code patterns, namely FPU context saving instructions and context switching sequences for syscalls. While these leaks do not seem to bother the execution of classic programs under DBI, at the price of an increased overhead a framework could be extended to patch them as soon as they become visible to user code[2].

Memory contents and permissions. Presenting the code under analysis with a faithful memory state as in a native execution is inherently difficult for a DBI system, as it operates at the same privilege level and in the same address space of the program. We identify three aspects that matter in how memory gets presented to the code under analysis.

The first aspect is correctness. As we have seen, memory permissions mirroring a native execution may not be enforced faithfully, resulting in possibly unfeasible executions. For instance, a DBI system may not detect when the virtual instruction pointer reaches a region that is not executable, continuing the instruction fetching process rather than triggering an exception. This may result in, e.g., reviving classic buffer overflow attacks as shown in [28]. Similarly, failing to reproduce guard page checks may be problematic when dealing with programs like JIT compilers that use them. Adding these checks may introduce unnecessary overhead for many analyzed applications, so they could be made available on demand.

The second aspect is escaping. When an adversary gets a pointer inside the code cache, injection attacks become a possibility. This problem is shared with general-purpose JIT engines [54], such as those used in browsers for running Javascript code efficiently. One possibility could be to use read-only code caches as in the mobile version of Safari, incurring a performance penalty by switching it to writable non-executable when modifications are made by the engine; during a transition however legit accesses from concurrent threads could fail, and proper handling would be needed.

Keeping thread-specific code caches mitigates the problem at the price of increased memory occupancy[3] and is available as an option for instance in DynamoRIO. This solution is however still subject to race conditions attacks as shown in [54]. The authors propose a mechanism where a writable code cache is moved to another process, and its pages shared with the process under analysis with read-only privileges. Their approach incurs a small performance overhead on Strata and on the V8 JavaScript engine.

[2] https://github.com/dynamorio/dynamorio/issues/698.

[3] This may however even let instrumentation code run faster when thread-specific.

The third aspect is evasion. Due to the shared address space, an adversary may simply register an exception handler and scan the memory looking for unexpected contents, such as the presence of the code cache or internal structures and code from the runtime. Shepherding all memory accesses would solve the evasion problem, but the overhead could increase dramatically [4].

One mitigation proposed in Strata is to use metamorphism to alter the contents of the code cache, hindering evasion and escape attacks based on pattern recognition. This could be paired with guarded pages and mechanisms to move the location of the code cache (and possibly internal structures) periodically during the execution, triggering however the invalidation of all compiled code as if the engine were detached and reattached to the process.

These countermeasures would still not be sufficient against attackers with code reading capabilities. Consider the case of a loop trying to read the page containing its own instructions in the cache: such an access cannot be denied on the x86 architecture. An answer to the problem may come from recent research on non-readable executable memory (XnR) [1] to prevent disclosure exploits for code reuse attacks. While the default XnR setting does not support our "in-page" reading loop scenario, follow-up techniques can be used to handle it: in particular, [62] shows how to achieve an effective separation between read and execution capabilities using Extended Page Tables on a thin hypervisor layer. Further hardware assistance could ease both the implementation and deployment effort: for instance, the ARMv8 processor provides facilities to support this mechanism at kernel level [62], while the Intel Skylake x86 architecture has introduced Memory Protection Keys to control memory permissions in user space that could be used to achieve executable-only memory as described in [33].

DBI engine internals. Changes to the execution context may be necessary for the sake of performance, for instance to keep internal data structures of the runtime quickly accessible using TLS mechanisms. An engine may attempt to randomize the TLS slot in use and hide its presence to queries from the application, but when an adversarial sequence tries to allocate all the slots the engine can either abort the execution (similarly as in what happens when the memory occupied by the engine prevents further heap allocation by the program) or resort to a less efficient storage mechanism.

The presence of trampolines on special DLL functions could be hidden using the same techniques for protecting the code cache, providing the original bytes expected in a read operation as in [46]. Write operations are instead more difficult, as the attacker may install a custom trampoline that either returns eventually to the original function (which still needs to be intercepted by the engine) or simply alter the standard semantics for the call in exotic ways.

Interactions with the OS. DBI frameworks can massage the parameters and output values of some library and system calls in order to achieve the design guideline G2, that is, hiding unavoidable changes from the program (Section 4.1). For instance, DynamoRIO intercepts memory query operations to its own regions to let the program think that such areas are free [4]. While allocations in such regions could be allowed in principle by moving the runtime in other portions of the address space, resource exhaustion attacks are still possible on 32-bit architectures, and are not limited to memory (for instance, file descriptors are another possibility). As system call interposition can incur well-known traps and pitfalls [18], DBI architects implement such strategies very carefully. Observe that remote memory modifications carried out from another process could be problematic as well, but a kernel module could be used to capture them [4].

Exception handling. A DBI system has to present a faithful context to the application in the presence of exceptions and signals. DBI architects are faced with different options in when (if an interruption can be delayed) and how the context translation has to take place; also, there are cases extreme enough for mainstream applications that can be handled loosely [4]. Systems like Valgrind that work like emulators by updating the virtual application state at every executed instruction are not affected by this problem. However, they incur a higher runtime overhead compared to others (e.g., Pin, DynamoRIO) that reconstruct the context only when needed.

Translation defects. Instruction errata and alike defects can be brought under two main categories: implementation gaps and design choices. Apart from challenging sequences such as 32-to-64 switches, errata from the first category can be addressed through implementation effort for code domains where they are problematic. On the other hand, defects may arise due to design choices aimed at supporting efficient execution of general programs: this is the case with self modifying code within the current basic block, which as we said can be detected in Pin with an optional switch at the price of an increased overhead.

To conclude our discussion, we would like to mention the possibility of having user-supplied analysis code pollute the transparency of the execution, for instance in its context reconstruction process. Frameworks like Pin that support registering analysis callbacks might be slightly easier to use for analysis writers compared to others like DynamoRIO that let users manipulate statements directly, but DBI systems may hardly avoid leaving part of the responsibility for transparency in the hands of their users.

5.2 Mitigations at Analysis Code Level

While revisiting architectural and implementation choices behind a DBI system can bring better transparency by design, some research has explored how user-provided analysis code can mitigate artifacts of mainstream DBI systems and defeat evasive attempts observed in some applications domains. This approach has been proposed by PinShield [46] and adopted in the context of executable unpacking in the presence of anti-instrumentation measures.

We have designed a high-level DBI library that could run in principle in existing analysis systems to detect and possibly counter DBI evasion and escape attempts. We revisited the design of PinShield to achieve better performance when shepherding memory accesses, and introduced protective measures to cope with memory permission consistency (e.g., to enforce NX policies and page guards), pointer leaks using FPU instructions, inconsistencies in exception handling, and a number of detection queries to the OS (e.g., for when the DBI engine is revealed a debugger).

For a prototype implementation we chose the most challenging setting for user-provided stopgap measures: we target the popular combination of Pin running on Windows. Unlike DynamoRIO, Pin does not rewrite the results of basic fingerprinting operations that can give away its presence like OS queries about memory. As it is

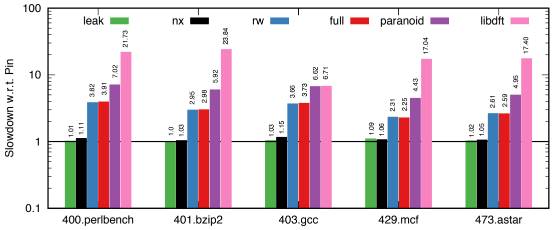

Figure 1: Performance impact of user-provided mitigations.

closed source, it cannot be inspected or modified to ease the implementation of mitigations, and using Windows makes immediate cooperation on the OS side (e.g., via a kernel driver) more difficult.

Approach. We pursued a design for the library that could be portable to other frameworks and Linux, avoiding uses of specific primitives or choices that could tie it to Pin’s underpinnings.

To shepherd memory accesses, we maintain a shadow page table as an array indexed by the page number for the address. We monitor basic RWX permissions and page guard options using 4 bits in a 1-byte element per page[4]. For 32-bit Windows, this yields a table of 512 kbytes for the 2-GB user address space, with recently accessed fragments likely to be found in the CPU cache for fast retrieval. We update the table in presence of code loading events and every time the program (or Windows components on its behalf) allocates, releases, or changes permissions for memory, hooking system calls and events that may cause such changes to the address space. When a violation is detected, we create an exception for the application[5].

Possible code pointer leaks from FPU instructions are intercepted as revealing instructions get executed: we replace the address from the code cache with the one in the original code. For this operation one can either resort to APIs possibly offered by an engine to convert addresses, or monitor the x87 instructions that cause the FPU instruction pointer to change with a shadow register. Although more expensive, we opted for the second approach for generality.

Exception handling inconsistencies may be unrelated to memory: this is the case with the single-step exception and `int 2d` attacks found in malware and executable protectors. We intercept such sequences and forge exceptions where needed.

Due to lack of space, we refer the reader to our source code for mitigations made of punctual countermeasures, such as pointer leaks with `int 2e` attacks and detections based on debugger objects.

Overhead. The mitigations presented above can have a significant impact on the baseline performance level offered by an engine running an empty analysis code. Shepherding memory accesses is a daunting prospect for DBI architects [4], let alone when implemented on top of the engine. However, it may be affordable for a user-defined analysis that already has to track such operations such as taint analysis. Similarly, conformance checking on NX policy slows down the execution as it requires that target of branches be checked, but may be acceptable for code that already validates transfers, for instance to support CFI or other ROP defenses.

We conducted a preliminary investigation on the SPEC CPU2006 benchmarks commonly used to analyze DBI systems [4, 34]. We consider different protection levels: pointer leaks, NX and page guard checks on indirect transfers, denying RW access to DBI regions, the three strategies together, and a paranoid variant[6]. We also consider a popular taint analysis library for byte-level tracking granularity in its default configuration with 1-byte tags. Due to lack of space (a more complete discussion is provided in Appendix A) we report figures for a subset of benchmarks in Figure 1.

Tracking x87 instructions for leaks has a rather limited impact on execution time. Enforcing NX and page guard protection on indirect transfers seems cheap as well. Shepherding memory read/write accesses incurs a high slowdown, but smaller than the one introduced by the heavy-duty analysis of libdft. High overheads were expected, but we do not find these figures demoralizing: while some performance can be squeezed by optimizing the integration with the backend and with analyses meant to run on top of it, we believe a fraction of this gap can be reduced if countermeasures get implemented inside the engine. We used evasive packer programs to stress the implementation, and we were able to run code packed with PELock that [46] based on PinShield could not handle properly.

Discussion. Mitigating artifacts using user-level code is a slippery road. As we mentioned in Section 5.1, there are aspects in system call interposition that if overlooked can lead security tools to easily be circumvented: one of them is incorrectly replicating OS semantics. We follow the recommendations from [18] by querying the OS to capture the effects of system calls that manipulate regions of the address space: this helped us in dealing with occasional inconsistencies between the arguments or output parameters for such calls and the effects observed on the address space. Our library pursues at analysis code level the DBI system design guideline G2 on making discrepancies imperceivable to the monitored program (Section 4.1), and its performance impact could be attenuated if cooperation occurs on the DBI runtime side, for instance by supporting automatic (optimized) guard insertion during trace compilation.

6 WRAP-UP AND CLOSING REMARKS

In the previous sections we have illustrated structural weaknesses of the DBI abstraction and its implementations when analyzing code in a software security setting, and discussed mitigations to make DBI frameworks more transparent—at the price of performance penalties—in the form of adjustments to their design or stopgap measures inside analysis tools. We conclude by discussing implications for researchers that want to use DBI for security with respect to instrumentation capabilities and to the relevance of the evasion and escape problems, putting into the equation the attacker’s capabilities and what is needed to counter adversarial sequences.

Choosing DBI in the first place. As we have seen in Section 3, the flexibility of DBI primitives has supported researchers in developing a great deal of analysis techniques over the years. Especially when the source code of a program is not available, there are essentially two options that could be explored other than DBI: SBI and VMI techniques. Although tempting to make a general statement on when one approach should be preferred, we believe the picture is not so simple, and thorough methodological and experimental comparisons would be required for different application domains.

[4] We leave 4 bits to encode more policies or host data from upper analysis layers.

[5] PinShield in such cases allows the access but rewires it to another region. We can still use their approach for instance to protect `ntdll` trampolines.

[6] For when ESP holds data or EIP flows into a page with inconsistent permissions via hard-wired jump offsets or branchless sequences that cross page boundaries [24].

We can however elaborate on three aspects that could affect a researcher's choice. The first aspect is related to the instrumentation capabilities of each technology. SBI can instrument a good deal of program behaviors as long as static inference of the necessary information is possible. On the other hand, VMI can capture generic events at whole-system level regardless of the structure of the code, using libraries that bridge the semantic gap to determine which events belong to the code under analysis. However, current VMI systems cannot make queries or execute operations using the APIs provided by the OS, unlike DBI systems that naturally let their users to. A 2015 work proposed with PEMU [65] a new design to move DBI instrumentation out of VM, providing a mechanism called forwarded guest syscall execution to mimic the normal functioning of a DBI engine; however, the public codebase the authors made available seems no longer under active development.

The second aspect is related to the deployability of the analysis system runtime. In the case of SBI, the requirements are typically limited as the original binary gets rewritten. For the DBI abstraction, the use of process virtualization paves the way for building tools that operate on a single application in a possibly lightweight manner, enabling their use also in production systems, e.g., when legacy binary are involved [4]. For VMI technology, bringing up an emulated or virtualized environment may very well be a daunting prospect or a natural choice depending on the application scenario.

The third aspect is related to whether the analysis should not only monitor, but also alter the execution when needed. To this end, DBI and VMI are both capable of detecting and altering specific instructions also when generated at run time. VMI technology is currently lackluster in aspects that involve replacing entire calls or sequences in the text of a program; on the other hand, system call authorization can be dealt with as context switches occur.

Dealing with adversarial code. A fourth aspect, possibly the most appealing for our readers, can be added to the technological discussion: adversarial sequences. We should distinguish between general detection techniques, which may affect more technologies at once, and ad-hoc detection patterns, sometimes for a specific runtime.

For example, code very sensitive to time variations is likely to deviate from the normal behavior when executing under DBI or other in-guest dynamic analysis systems: consider for instance time-based anti-analysis strategies in evasive malware. On the other hand, code checksumming sequences give away several analysis systems, but DBI ones are normally not bothered by them.

A crucial issue is related to the characteristics of the code that undergoes analysis, that is, if there is the possibility for an attacker to embed adversarial patterns specific to DBI. This is particularly (but not only) the case of research focused on malicious code analysis and reverse engineering activities. Such code can clearly challenge massaged results and other mitigations for transparency issues put in place by the engine or the analysis tool.

When the adversary does not have arbitrary memory read capabilities, mitigating leaked code pointers is already sufficient in most cases to hide the presence of extra code regions, and different attack surfaces should be tried, such as exhaustion or prediction attacks for memory allocations. When the adversary cannot write to arbitrary memory locations and leaked pointers have already been dealt with, escaping attempts are essentially contained, while the execution of unfeasible flows from inconsistencies in enforcing NX policies can be avoided by shepherding control transfers.

What really raises the bar for DBI engines is coping with an attacker that can register exception handlers and force memory operations also on regions marked as unallocated in massaged OS queries. The runtime overhead of shepherding every memory access is intrinsically high for current DBI designs, but technologies like executable-only memory may come to the rescue in the future.

One may also wonder whether the popularity of an analysis technique can eventually lead to the diffusion of ad-hoc adversarial sequences in the code it is meant to analyze. In the history of DBI we are aware of ad-hoc evasions against Pin in some executable packers [10], as simple DBI-based unpacking schemes from a few years ago were effective against older generations of packers. On the contrary, there is a possibility that even when an attack surface is well known and not particularly hard to exploit, as in the case of implicit flows against taint analysis [7], adversaries may focus their attention elsewhere for anti-analysis sequences: for the proposed example, malware authors in late years seem rather to have concentrated their efforts on escaping sandboxing technologies and hindering code analysis with obfuscation strategies such as opaque predicates and virtualization. We thus find it hard to speculate on an arms race that could involve DBI evasion in the near future.

Finally, as we mentioned in Section 4.1 also approaches like VMI that are more transparent by design can become fragile in the presence of a dedicated adversary. Every technology has its own trade-offs between transparency and aspects like performance or instrumentation capabilities. In the case of SBI, the approach can offer better performance than DBI, but its transparency is more fragile than DBI and VMI: for instance, even recent SBI embodiments cannot offer protection against introspective sequences when these are not identified in the preliminary static code analysis phase.

Closing remarks. In this work we attempted to systematize existing knowledge and speculate on a problem that recently received a good deal of attention in the security community, with opinions sometimes at odds with one another. We think that using DBI for security is not a black and white world, but is more about how the DBI abstraction is used once the user is aware of its characteristics in terms of execution transparency, understood not only as a security problem, but sometimes also as a correctness one. Hoping that other researchers may benefit from it, we make available our library of detection patterns and mitigations for Pin at:

https://github.com/season-lab/sok-dbi-security/

Acknowledgements. We are grateful to Ke Sun, Xiaoning Li, and Ya Ou for sharing their code [56] with us. This work is supported in part by a grant of the Italian Presidency of the Council of Ministers.

REFERENCES

[1] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code *(CCS '14)*. ACM.

[2] Otto Brechelmacher, Willibald Krenn, and Thorsten Tarrach. 2018. A Vision for Enhancing Security of Cryptography in Executables *(ESSoS '18)*. Springer.

[3] Michael Brengel, Michael Backes, and Christian Rossow. 2016. Detecting Hardware-Assisted Virtualization *(DIMVA '16)*. Springer.

[4] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation *(VEE '12)*. ACM.

[5] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (Nov. 2000).

[6] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-engineering *(CCS '09)*. ACM.

[7] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment *(DIMVA '08)*. Springer.

[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing *(SP '15)*. IEEE.

[9] Yue Chen, Mustakimur Khandaker, and Zhi Wang. 2017. Pinpointing Vulnerabilities *(ASIA CCS '17)*. ACM.

[10] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 2018. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost *(CCS '18)*. ACM.

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. 1998. FX!32: A Profile-Directed Binary Translator. *IEEE Micro* 18, 2 (March 1998).

[12] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks *(ASIA CCS '11)*. ACM.

[13] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions *(CCS '08)*. ACM.

[14] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. 2011. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection *(SP '11)*. IEEE.

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

[16] S. J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy. 1990. Techniques for Efficient Inline Tracing on a Shared-memory Multiprocessor *(SIGMETRICS '90)*. ACM.

[17] Francisco Falcón and Nahuel Riva. 2012. Dynamic Binary Instrumentation Frameworks: I know you're there spying on me. *Recon* (2012).

[18] Tal Garfinkel. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools *(NDSS '03)*.

[19] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities *(HOTOS'07)*. USENIX Association.

[20] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection *(NDSS '03)*.

[21] Cosmin Gorgovan. 2014. Escaping DynamoRIO and Pin. https://github.com/lgeek/dynamorio_pin_escape.

[22] Sean Heelan and Agustin Gianni. 2012. Augmenting Vulnerability Analysis of Binary Code *(ACSAC '12)*. ACM.

[23] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'D My Gadgets Go? *(SP '12)*. IEEE.

[24] Martin Hron and Jakub Jermář. 2014. SafeMachine: Malware Needs Love, Too. *Virus Bulletin* (2014).

[25] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking *(CCS '13)*. ACM.

[26] Kangkook Jee, Georgios Portokalidis, Vasileios P Kemerlis, Soumyadeep Ghosh, David I August, and Angelos D Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware *(NDSS '12)*.

[27] Karl Trygve Kalleberg and Ole André Vadla Ravnås. 2016. Testing interoperability with closed-source software through scriptable diplomacy. (FOSDEM '16).

[28] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. 2018. PwIN – Pwning Intel piN: Why DBI is Unsuitable for Security Applications *(ESORICS '18)*. Springer.

[29] James R. Larus and Eric Schnarr. 1995. EEL: Machine-independent Executable Editing *(PLDI '95)*. ACM.

[30] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces *(CCS '18)*. ACM.

[31] Xiaoning Li and Kang Li. 2014. Defeating the Transparency Features of Dynamic Binary Instrumentation. *BlackHat USA* (2014).

[32] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution *(NDSS '08)*.

[33] Daiping Liu, Mingwei Zhang, and Ravi Sahita. 2018. XOM-switch: Hiding Your Code from Advanced Code Reuse Attacks In One Shot. *BlackHat ASIA* (2018).

[34] 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 *(PLDI '05)*. ACM.

[35] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. 2010. Conqueror: Tamper-Proof Code Execution on Legacy Systems *(DIMVA '10)*. Springer.

[36] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code *(CCS '15)*. ACM.

[37] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. PatchDroid: Scalable Third-party Security Patches for Android Devices *(ACSAC '13)*. ACM.

[38] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs *(CC '02)*. Springer.

[39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation *(PLDI '07)*. ACM.

[40] Mathias Payer and Thomas R. Gross. 2011. Fine-grained User-space Security Through Virtualization *(VEE '11)*. ACM.

[41] M. Payer and T. R. Gross. 2013. Hot-patching a web server: A case study of ASAP code repair *(ACSAC '13)*. IEEE.

[42] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers *(EUROSEC '11)*. ACM.

[43] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: Force-executing Binary Programs for Security Applications *(SEC'14)*. USENIX Association.

[44] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2015. DynaGuard: Armoring Canary-based Protections Against Brute-force Attacks *(ACSAC '15)*. ACM.

[45] Jonas Pfoh and Sebastian Vogl. 2017. rVMI: A New Paradigm for Full System Analysis. *Black Hat USA* (2017).

[46] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware *(DIMVA '17)*. Springer.

[47] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. *(NDSS '15)*.

[48] Quarkslab. 2019. QDBI. https://qbdi.quarkslab.com/.

[49] Nguyen Anh Quynh. 2018. SKORPIO: Advanced Binary Instrumentation Framework. (OPCDE '18).

[50] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and Optimization of Win32/Intel Executables Using Etch *(NT'97)*. USENIX Association.

[51] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. 2003. Retargetable and Reconfigurable Software Dynamic Translation *(CGO '03)*. IEEE.

[52] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. 2010. Dynamic program analysis of Microsoft Windows applications *(ISPASS '10)*. IEEE.

[53] Asia Slowinska, Istvan Haller, Andrei Bacs, Silviu Baranga, and Herbert Bos. 2014. Data Structure Archaeology: Scrape Away the Dirt and Glue Back the Pieces! *(DIMVA '14)*. Springer.

[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2003. Exploiting and Protecting Dynamic Code Generation *(NDSS '15)*.

[55] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools *(PLDI '94)*. ACM.

[56] Ke Sun, Xiaoning Li, and Ya Ou. 2016. Break Out of the Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. *Black Hat Asia* (2016).

[57] Cisco Talos. 2017. PyREBox. https://github.com/Cisco-Talos/pyrebox.

[58] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. 2008. Towards Automatic Reverse Engineering of Software Security Configurations *(CCS '08)*. ACM.

[59] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and Emergency Patch Generation for Integer Overflow Exploits *(DIMVA '14)*. Springer.

[60] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. Behavior Based Software Theft Detection *(CCS '09)*. ACM.

[61] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao. 2018. To Detect Stack Buffer Overflow with Polymorphic Canaries *(DSN '18)*. IEEE.

[62] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software *(ASIA CCS '16)*. ACM.

[63] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves *(CCS '17)*. ACM.

[64] Guanhua Yan, Nathan Brown, and Deguang Kong. 2013. Exploring Discriminatory Features for Automated Malware Classification *(DIMVA'13)*. Springer.

[65] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. 2015. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework *(VEE '15)*. ACM.

[66] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A Platform for Secure Static Binary Instrumentation *(VEE '14)*. ACM.

[67] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen, and Michael D. Smith. 1997. System Support for Automatic Profiling and Optimization *(SOSP '97)*. ACM.

[68] Jingyu Zhou and Giovanni Vigna. 2004. Detecting Attacks That Exploit Application-Logic Errors Through Application-Level Auditing *(ACSAC '04)*. IEEE.

A COMPLETE EXPERIMENTAL RESULTS

In the following we describe the experimental setup used for a preliminary assessment of the performance impact of the stopgap measures we implemented as a high-level library for analysis tools, and present complete results for the corpus of C/C++ benchmarks in the SPEC CPU2006 suite (Table 2) that we used to measure it.

We use Pin 3.5 running on a Windows 7 SP1 32-bit machine with negligible background activity. For each experiment we use a clean virtual machine with 1 CPU core and 3 GB of RAM hosted on VirtualBox 5.2.6 running on a server with Debian 9.6 Stretch and two Intel Xeon E5-4610 processors. Both the library and the benchmarks are compiled using Visual Studio 2010 in Release configuration. We consider the average value for 9 trials of each configuration.

For integer benchmarks we can see from Figure 2 that the impact of the mitigation for pointer leaks is small, with a slowdown in the worst case as high as 1.09x compared to an execution under Pin with no instrumentation and analysis code. The overhead originates from spilling the x87 instruction pointer to a tool register at every non-control FPU instruction, with the JIT compiler of Pin taking care of register reallocation. The impact of the NX mitigation, which protects instruction fetching from non-executable or guarded[7] pages, is intuitively related to the number of indirect transfers (call, ret, and jump operations) in the program, with a peak of 1.17x for benchmark omnetpp (417). The overhead from shepherding every read/write memory access to protect the code cache and other regions of the runtime ranges from 2.31x to 5.02x.

Combining the three mitigations in the full configuration incurs an overhead very similar to the one from RW, which unsurprisingly is by far the most expensive on these benchmarks. However in some cases the overhead of full might be slightly lower than having the sole RW mitigation active. Once we ruled out noise in time measurements, we believe the cause might lie in how the quality of the JIT-ted code is affected by the guards inserted for NX and RW as well as by the register re-allocation caused by spilling tool registers in the implementation of the leak and RW mitigations.

The paranoid mode is clearly more expensive as it shepherds every instruction fetch to enforce NX and page guard protection on code that crosses the border of two pages with different permissions with no intervening jumps, and verifies read/write permissions for push and pop operations for when the stack pointer may be used as a general-purpose register.

For floating-point benchmarks we can observe similar trends as for the integer ones with two notable exceptions: namd (444) and sphinx3 (482). We can see that for these two benchmarks the NX mitigation incurs a overhead considerably higher than in the other programs from the CINT and CFP sets. A first inspection of the execution profiles seems to suggest that both programs make a high usage of indirect control transfers, and the insertion of guards for such transfers may affect the trace linking process inside Pin or other aspects of the JIT compilation process. Unfortunately we could not verify this claim by monitoring the code generation process due to the closed-source nature of Pin.

Suite	ID	Name
CINT	400	perlbench
	401	bzip2
	403	gcc
	429	mcf
	445	gobmk
	456	hmmer
	458	sjeng
	464	h264ref
	471	omnetpp
	473	astar
CFP	433	milc
	444	namd
	450	soplex
	453	povray
	470	lbm
	482	sphinx3

Table 2: Integer and floating-point C/C++ benchmarks.

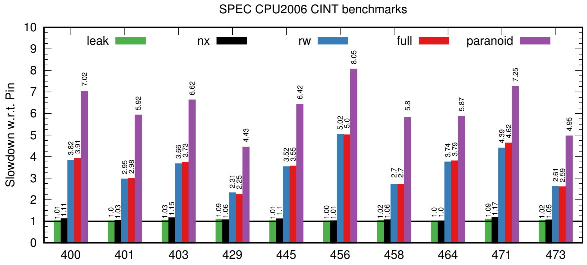

Figure 2: Slowdown for integer benchmarks.

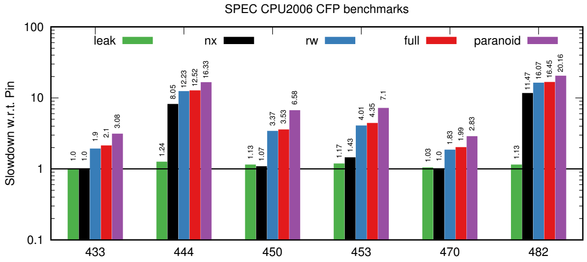

Figure 3: Slowdown for floating-point benchmarks.

Compared to the original implementation of PinShield[8], the RW mitigation from our library was at least one order of magnitude faster in the tests we conducted: while the guards we insert incur a O(1) lookup inside an array, PinShield scans a linked list of known memory ranges for every memory read or write access. As we mentioned in Section 5.2, pointer leaks via x87 instructions and NX/page guard conformance checking are not handled by PinShield.

[7] Pin by design already handles page guards for read and write operations correctly.

[8] https://github.com/Phat3/PINdemonium/.