Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts

DANIELE CONO D'ELIA, Sapienza University of Rome, Italy
LORENZO INVIDIA, Sapienza University of Rome, Italy
FEDERICO PALMARO, Prisma S.r.l., Italy
LEONARDO QUERZONI, Sapienza University of Rome, Italy

Dynamic binary instrumentation (DBI) systems are a popular solution for prototyping heterogeneous program analyses and monitoring tools. Several works from academic and practitioner venues have questioned the transparency of DBI systems, with anti-analysis detection sequences being found already in malware and executable protectors. The present Field Note details new and established detection methods and evaluates recent versions of popular DBI systems against them. It also sets out reflections on potential remediations and alternatives available to security researchers for their daily needs. We make available a large collection of implemented detections, hoping it can help the community build better DBI runtimes and tools.

CCS Concepts: • **Security and privacy → Software reverse engineering**; *Systems security*.

Additional Key Words and Phrases: anti-analysis, binary analysis, dynamic binary instrumentation, evasion, malware, packers

ACM Reference Format:
Daniele Cono D'Elia, Lorenzo Invidia, Federico Palmaro, and Leonardo Querzoni. 2021. Evaluating Dynamic Binary Instrumentation Systems for Conspicuous Features and Artifacts. *Digit. Threat. Res. Pract.* 1, 1, Article 1 (January 2021), 12 pages. https://doi.org/10.1145/3478520

1 INTRODUCTION

Dynamic binary instrumentation (DBI) is an execution paradigm that enables the insertion of probes and analysis callbacks in an executable program while it is running. Instrumentation can serve heterogeneous execution monitoring and altering purposes ranging from coarse-grained execution profiling to enforcing software fault isolation schemes. DBI techniques have received a great deal of popularity across different research communities, especially in the programming language, software security, and systems domains. They are well supported by industry, with companies of the likes of Intel and Google that actively develop DBI systems for the community.

The popularity of DBI comes from several factors. First and foremost, easy-to-use instrumentation: to add a monitoring sequence the user need not worry about context saving and binary code manipulation, but only registers callbacks for classes of instructions (e.g., control transfer, data movements). The engine abstracts away the idiosyncrasies of a specific architecture and compiles efficient analysis code for it. At the end of the day, researchers in most cases do not need to be DBI experts to be able to write a DBI tool that embodies their idea [9].

A much-debated aspect for DBI is, however, its transparency. This is particularly relevant for security scenarios like malware analysis, where a dedicated adversary may recognize the presence of a monitoring system and adapt

Authors' addresses: D.C. D'Elia, L. Invidia, and L. Querzoni, Department of Computer, Control and Management Engineering "Antonio Ruberti", Sapienza University of Rome, Via Ariosto 25, 00185 Rome, Italy; F. Palmaro, Prisma S.r.l., Via Mario Bianchini 51, 00142 Rome, Italy. Correspondence: delia@diag.uniroma1.it.

2576-5337/2021/1-ART1
https://doi.org/10.1145/3478520

their tactics, undermining the soundness or completeness of an analysis. Over the last decade, several works from practitioner [13, 19, 23, 35] and academic [9, 10, 14, 22, 30] venues have presented detection methods for DBI. A fragmented view presents to users: the efficacy of a method may change for systems different to the one analyzed, but also across releases of the same engine since DBI systems are under constant development. Researchers have also come to different conclusions regarding the suitability of using certain DBI systems for security research.

Contributions. Building on our experience in developing DBI tools for malware analysis [10, 11], in this Field Note we systematize detection methods for DBI and evaluate in depth the two reference systems in the community: Pin and DynamoRIO. We also cover in brief three systems that, in spite of possibly more limited compatibility or flexibility, are now trending in a few research areas: Dyninst, QBDI, and QEMU in User Mode emulation.

In our study, we analyze techniques specific to DBI and expand the scope to adversarial sequences developed in other domains that turn out to be effective against DBI too. We also present several original detection methods. We conclude by sharing our reflections on the practical relevance of DBI detections, on research directions for practical remediations, and on emerging alternate instrumentation technologies.

Hoping to foster further research in the area, we share with the community a collection of DBI detections with all the implementations that we exercised. The code is available at:

`https://www.github.com/dcdelia/dbi-detector/`

2 CONSPICUOUS FEATURES AND THEIR DETECTION

A recent work [9] observes that there are two connotations of transparency that pertain to DBI systems. For a DBI architect, transparency means ensuring that a program executes exactly as in an uninstrumented native environment, preserving interoperability with other applications[1]. For a security researcher, transparency also implies the possibility of an adversary looking for anomalous features of the execution environment: while those do not alter the semantics of normal programs, they are however amenable to detection attempts.

To better understand why transparency issues arise in the first place, we provide a brief overview of how the majority of modern DBI systems work. A DBI engine operates on binary, user-space code and wraps its execution in a process-level virtualization fashion: the engine does not execute the original instructions directly; instead, they undergo a dynamic compilation process. A just-in-time (JIT) compiler analyzes, instruments, and encodes in a directly executable form the original program instructions as they are about to execute. The JIT operates at the granularity of traces, which end up populating a code cache used to avoid frequent recompilations and ameliorate performance. Further code optimizations such as trace linking are then possible inside the runtime.

Most notably, the engine strives to provide *architectural state transparency*: it rewrites the output of introspective actions to present the program with the same addresses and data observable in a native execution. This implies that for instance, whenever the program inspects the instruction pointer, the engine should expose the `.text` address corresponding to the JIT-ed instruction (real EIP) that is currently executing from the code cache.

The Roots of All Evil. In this characteristic design we can point out three main sources of concern for transparency:

- **shared address space**: the application under analysis and the DBI engine operate in the same address space and at the same privilege level, which opens the door to detection and even subversion attempts [9];
- **contention for operating system resources**: the application and the engine have access to the same set of resources, such as file descriptors, memory mapped files, physical memory, signals, and so on;
- **overhead from JIT compilation**: when compared to a native execution, the temporal behavior of a code portion is subject to slowdowns, but also to variability over time from dynamic (re)compilation choices.

While sharing the address space and OS resources is an intrinsic limitation of DBI from a security standpoint, at the same time it represents an advantage when implementing a research tool. For instance, a DBI tool can

[1] An excellent treatment of transparency aspects in the design of a DBI system can be found in [6].

directly query the OS in order to retrieve the list of open handles or allocated heap regions at a given time, while a tool operating e.g. from a virtual machine monitor would need an arguably more complex inspection procedure.

In addition to the three sources listed above, **implementation gaps** are a recurrent issue for systems based on binary recompilation or translation: missing support or imprecise handling of rare instructions and constructs is common in practice and can expose the presence of a DBI system to an adversary.

Building on these considerations, we identify four aspects of program execution under DBI that are directly influenced by these sources: *memory consistency*, *process behavior*, *temporal behavior*, and *translation defects*. For a detection sequence it is then typically sufficient to target one of these aspects to succeed.

The categorization that we use in this work (Table 1-2-3) accounts also for two factors involving the generality of the method: (i) use of primitives that are exclusive of a specific execution platform, i.e., the operating system that runs the DBI engine and the analyzed program; and (ii) efficacy across DBI systems, for detections stemming from reverse engineering work carried on a specific system looking for its distinctive fingerprints. The categorization aims at capturing how architectural traits of DBI systems impact execution. Those traits are also intimately tied to deployment (e.g., interactions with the OS) and engineering (e.g., application state transparency vs. run-time overhead trade-offs [6]) choices when implementing a DBI runtime. Categorizations from prior literature [9, 22, 30, 33] approach detections from other angles (focusing on, e.g., the point of view of an adversary in [9] or the component in the DBI runtime to target for detection in [22]), and may be of independent interest to our readers.

Categories for Artifacts. We now discuss how the DBI engine design inevitably impacts these four execution aspects, referring our readers to [6, 9] for more technical insights. Let alone the presence of extra regions used by the DBI runtime in the address space, perfect **memory consistency** is difficult to preserve especially for performance reasons. For instance, when some code is located in an executable page and the JIT engine compiles traces for it, an adversarial program in the meantime can change the page permissions and execute the code again. A native CPU would detect the violation thanks to hardware assistance for memory; a DBI engine usually avoids such check, which is expensive when realized via software, and simply runs the previously JIT-ed code for it. The engine also assumes that the code layout is well-behaved: for instance, when a basic block spans two pages, the engine may not check the permissions of the second page if the first is executable.

Process behavior is inevitably affected by the actions of the DBI engine. We can observe contention for resources provided by the OS in limited numbers (such as the handles that a process can open or its thread-local storage slots), and different levels of pressure on the OS for hosting the process natively than under DBI (such as the virtual memory peak or the resident set size). We speculate that even an engine possibly running in a separate process and using shared memories for exposing JIT-ed code may ameliorate but not solve this problem.

This becomes more evident for **temporal behavior**. JIT-ed code that executes on the CPU is different (by a small or large extent depending on the instrumentation from the user's DBI tool) to the one present in the program. Even when the warm-up stage of the code cache ends there may still be subtle differences in the execution time for a given code fragment. DBI architects often tackle timing transparency by trying to ensure that timing differences are still a valid, yet infrequent, behavior observable on the native hardware [6]. However, the possible presence of a dedicated adversary inevitably changes the picture and undermines this assumption.

Finally, **translation defects** originate from an implementation gap or an efficiency choice. The former happens to be the case with rare instructions or with patterns intrinsically complex to handle for dynamic translators, such as a far return instruction to a different code segment. The latter instead happens when for performance the engine sacrifices full transparency of the architectural state and/or makes assumptions on the "well-behaving" of the program. For instance, engines may not mask the FPU instruction pointer value when executing x87 instructions, revealing a code cache address upon inspection. Self-modifying code (SMC) is more challenging: while an x86 CPU would automatically detect and execute the new code, a DBI engine needs to intercept such modifications to the code, rewire the involved program addresses, and update its code cache accordingly.

Detection of Artifacts. A dedicated adversary can then take advantage of the shared address space and identical privilege level to embed detection sequences in the execution. The presence of DBI in many scenarios is a clear indicator that a program is running outside its designated target. For instance, malware writers can make their samples show a benign behavior or terminate prematurely to resist analysis. Similarly, software vendors want to protect their programs as much as possible from intellectual property theft or piracy attempts, which need reverse engineering work: software protectors like Obsidium and PELock, destined for vendors but also abused by malware writers, insert detection sequences that specifically target DBI (e.g., exposing the code cache in PELock). In other scenarios like software fault isolation, Kirsch et al. [22] observe that the presence of DBI allows a threat actor to refine their exploitation strategy and subvert the very DBI engine to realize their goals.

We have surveyed and analyzed detection primitives for DBI presented in the literature over the last decade [9, 10, 13, 14, 19, 22, 23, 30, 35]. We report the primitives in Table 1-2-3 and detail some of them throughout Section 3. When more works claim a technique as new we credit the authors that reported it first.

We then experimented with anti-debugging techniques. We scrutinized collections of debugger detections like [5, 29] that are popular among practitioners, identifying conceptually different techniques and selecting for our tests multiple variations of a single theme when applicable (e.g., with single-stepping sequences). We then added to the testing suite lesser-known detections that we could find on specialized websites and blogs [1–3, 21, 26].

Overall, we used over a hundred detections detailed over the years by Ange Albertini, Walied Assar, Gabriel Barbosa, Rodrigo Branco, Peter Ferrie, Christopher Kannen, Pedro Neto, Tomislav Pericin, and Daniel Plohmann. Those should hopefully cover a significant fraction of an historically wealthy literature. To our surprise, several of these methods revealed DBI out of the box or after some adaptations, but no one reported them before as such. We mark these detections as *dbg* in the tables and reference the original source in the related discussions.

Finally, we introduce several novel detection primitives drawing from our experience with DBI. These are clearly marked as *new* in the tables.

3 A CLOSER LOOK AT POPULAR DBI SYSTEMS

Another factor that motivated our systematization work has been the natural evolution in the development of DBI frameworks in relation to when a detection method was first reported. For users, it may be unclear whether a detection method is still effective or a translation defect is still present, and if they should be concerned for their application scenario. Excluding [22], no notable work has exercised detection techniques on recent versions of DBI engines. For our evaluation we consider Pin [34] and DynamoRIO [6]: historically, these are the two best-performing and more actively developed DBI systems, and for this reason have been the main subject of prior DBI detection literature. We pick their latest version available (Pin 3.17, DynamoRIO 8.0.0) and run them on Windows 7 SP1, 8.1 & 10 v2004, and Ubuntu 18.04 (kernel 5.4.0-53) & 20.04 (kernel 5.4.0-48).

We would like to stress the importance of engine evolution with two anecdotes. Dynamic translators have historically struggled with self-modifying code (SMC). For instance, the popular QEMU system emulator needed several patches to handle the SMC sequences of the Windows PatchGuard technology. When years ago DBI was trending for automatic malware unpacking, packers such as tElock and PESpin shipped tricky SMC patterns that could crash or mislead DBI engines. SMC today remains abundant in malware [30], but Pin and DynamoRIO have nowadays significantly improved in the handling of it and become more difficult to expose[2].

Another prototypical detection method seen in malware and protectors (e.g., PELock) involves inspecting the instruction pointer via FPU instructions (Section 2): while Pin remains vulnerable, the developers of DynamoRIO have analyzed the performance implications from masking the real EIP value and remediated their handling[3].

[2]Pin natively handles SMC when the modification is on a basic block other than the one currently executing, and has a `-smc_strict` mode to handle in-block SMC with a performance penalty. DynamoRIO instead uses a write-only code cache: it intercepts both types of SMC, but on Windows its handler seems to assume that the program does not contain data beyond the top of the stack and pollutes several such locations.
[3]Part of the discussion is available at https://github.com/DynamoRIO/dynamorio/issues/698.

In the remainder of this section we present new detections and revisit several known ones, with special attention to those that have been affected by changes in DBI engines. Due to space limitations we leave out well-known, working detections not impacted by changes, and refer our readers to the research works reporting them and to our implementation for the associated detection code.

We also complement our evaluation with a brief review of detections that we attempted on three other systems. Albeit characterized by lower compatibility and maturity—or by more limited instrumentation primitives—than Pin and DynamoRIO, they have lately become a popular choice in some research areas: we consider the latest version available of Dyninst (11.0.0), QBDI (0.8.0), and QEMU (6.0.0-rc3) in User Mode emulation. We leave out instead publicly available systems that are tuned for specific capabilities (such as Valgrind [25] with its shadow values feature for memory-related analyses) or that saw fewer developments lately (such as libdetox [28]).

3.1 General Methods

Table 1 details 23 methods (4 new) agnostic to both the DBI engine and the execution platform.

The first new conspicuous feature that we found is that the image loading process of the engine may alter [3] pages that are amenable to *copy-on-write* optimization by the OS. For instance, Windows for efficiency tries to share memory for mapped files and section views between processes. We crafted an executable with a RWX code section that the native Windows loader loads and maps to pages with `PAGE_WRITECOPY_EXECUTE` permissions. We observe that Pin preserves the integrity of the loading process, while under DynamoRIO an introspection attempt from the process reveals `PAGE_EXECUTE_READWRITE` permissions for the pages instead.

I/O bytes are another conspicuous feature that to the best of our knowledge no one reported before. We use OS primitives to monitor the number of bytes read from storage by a process. We register higher values as the process retrieves module files for the DBI runtime. Then the engine in turn may access other files: for instance, Pin on Linux retrieves `/proc/self/cmdline`, while DynamoRIO periodically reads from `/proc/self/maps`.

With a similar approach, we can observe a higher *resident set size* (RSS) for an attacker-crafted program under DBI. On Linux special file `/proc/self/smaps_rollup` provides easy-to-parse tagged information for resident sets. As an example, for a small program that natively uses a RSS of 1404 kB (117 kB for private pages) we observe a RSS of ~17 MB (private: ~16 MB) under Pin and ~4 MB (private: ~2.8 MB) under DynamoRIO. The extra private pages are nearly equally split between anonymous and file pages for both engines. On Windows using `GetProcessMemoryInfo` we similarly observe abundant additional pages kept in RAM when under DBI.

Prefix decoding defects can expose an engine when combined with exceptional control flow [26]. We discovered that Pin crashes with an assertion failure when the single-byte `icebp` instruction[4] comes with a prefix. In general, adversaries can exercise exotic patterns that put pressure on the instruction decoding stage of the engine.

As for detections proposed in prior works, we observed that most of them work out of the box as expected also on the engine not targeted by the authors. For the *RWX page count* detection proposed for Pin in [13] we had to adjust the threshold as DynamoRIO exposes significantly fewer RWX pages to user queries. We also generalized the *trampolines* detection of [13] to check the integrity of the prologue of loaded OS APIs: while Pin tampers with 4 low-level Windows `ntdll` functions for the engine's internal working [34], DynamoRIO does it with 33.

We observe that engines still struggle when a *page policy change* affects code that the JIT compiler has recently processed. Pin and DynamoRIO may fetch and run a dynamically compiled version produced when the original code had legitimate permissions, missing that it now is in an inaccessible or guarded page, or even a non-executable one in the *W⊕X violation* case. Even more than a security issue, this is an execution correctness problem that grips DBI engines. As a remedy, interposing on page permission changes and invalidating the

[4]Formally `int1`, `icebp` is a breakpoint instruction that does not set any DR6 bit and triggers a #DB exception without checking for user/kernel privileges (hence this behavior also challenges hypervisors). Known for over 30 years, Intel acknowledged and documented it only in 2018.

Table 1. General detection methods for DBI engines. A checkmark indicates that the technique reveals the DBI engine. We mark novel detections as *dbg* when we borrowed and adapted them from anti-debugging literature, and *new* when original.

	Detection	Description	From	Pin	DRio
Memory consistency	Additional regions	Memory used by code modules loaded by DBI runtime (oblivious scan in [23])	[13]	✓	✓
	Adjacent pages	Permissions not reflected for code spanning pages with different permissions	[19]	✓	✓
	Copy-on-write	Engine touches program code pages loaded as copy-on-write by the OS	dbg		✓
	Egg hunting	Unique code sequence sees multiple occurrences (`.text` and code cache)	[35]	✓	✓
	Page policy change	Page permissions updates (becoming inaccessible or guarded) not reflected	[13, 19]	✓	✓
	RWX page count	Abnormal presence of pages with RWX permissions (JIT, code cache)	[13]	✓	✓
	Trampolines	Presence of hooking instructions that the engine inserts in low-level OS APIs	[13]	✓	✓
	W⊕X violation	Fetch and execute code from a non-executable page	[19]	✓	✓
Process behavior	API call count	Hook and count invocations of OS APIs that the JIT may use (e.g., allocation)	[13]		
	Available handles	Program can open fewer handles than the expected OS limit	[23]	✓	✓
	I/O bytes	Increase in statistics for read I/O bytes	new	✓	✓
	Parent process	Engine appears as parent of the program in the process hierarchy	[13]	✓	✓
	Resident set size	Higher RSS value as the OS keeps many pages in RAM for the DBI runtime	new	✓	✓
	VM peak	Higher VM peak value for resource usage by the process	[23]	✓	✓
Time	High latency	Code sequence takes longer to execute than with native instructions	[13]	✓	✓
	Module load latency	Latency of dynamic library loading (or reloading [22])	[13]	✓	✓
	Variable latency	Code sequence sees different latencies over time due to JIT effects (e.g., linking)	[22]	✓	✓
Translation defects	FPU context	Leak real EIP from FPU context using {`fstenv`, `fnstenv`, `fsave`, `fxsave`}	[13]	✓	
	Prefix decoding	Broken handling of prefix used for sensitive instruction (e.g., `repz fs icebp`)	dbg	✓	
	SMC in-block	Self-modifying code within a basic block (check if new or old code executes)	[30]	✓	
	SMC write-protect	SMC handling by on-write exception pollutes data written beyond top of stack	[19]		✓
	Trap flag	Set CPU TF=1 with `pushf+popf` and check if exception handler gets invoked	[10]	✓	✓
	Xmode code	Unsupported switch between 32 and 64-bit code (Heaven's gate and back)	[35]	✓	✓

involved code cache portions could help security users willing to pay a performance penalty in their tool. Similar correctness problems come from the *adjacent pages* detection of [19], not discussed in many subsequent works.

Table 1 lists also an obsolete method: *API call count*. For an adversary, it was possible to detect Pin by locating the `ZwAllocateVirtualMemory` Windows API in program memory and placing a hook in its code, so as to track any invocations made by the JIT compiler of the DBI runtime. Those are no longer visible in the Pin 3.x releases.

3.2 OS-specific Methods

Table 2 details 12 methods (3 new) that, in order to expose a generic DBI engine, make use of primitives or features that are characteristic of an OS and may not be available on others.

On Linux, the *stat EIP* detection accesses the special file `/proc/self/stat`, which exposes status information about the current process, to read the instruction pointer value (`kstkeip` field) for the process, revealing a code cache address for EIP. The other new detection involves *syscall tracing* using `ptrace`, a popular feature for program tracing. We fork a child process that a DBI engine normally follows and instruments as well. The child opts in for tracing with `PTRACE_TRACEME`, then the parent attaches to it. Under DynamoRIO the parent can observe a noisy syscall activity from the DBI runtime running in the child, while Pin seems able to conceal it.

On Windows, the *HW counting* detection [5] uses an OS-specific feature to set the debug registers of the CPU from unprivileged code. To this end, we acquire the CPU context and overwrite its DR0-3 registers (we can use `NtGetContextThread` or cause an exception and access it from an exception handler). Then in DR0-3 we set hardware breakpoints for a number (up to 4) of program locations in `.text` and make our program hit them a predetermined amount of times. We also set a handler for the single-step exception triggered by hardware breakpoints: this handler updates the hit count. Pin seems unaware of the presence of hardware breakpoints, so

Table 2. OS-specific detection methods for DBI engines.

	OS	Detection	Description	From	Pin	DRio
Proc. behavior	*Linux*	Signal mask	Engine catches signals not masked by program (`/proc/self/status`)	[23]	✓	✓
		`stat` EIP	Check EIP value from `/proc/self/stat`	new	✓	✓
		Syscall tracing	Spawn child process and detect with `ptrace` syscalls from engine in it	new		✓
	Windows	NoDebugInherit	Check EPROCESS field value with NtQueryIP(ProcessDebugFlags)	[10]	✓	
		TLS slots	Engine uses thread-local storage slots and the program inspects them	[35]	✓	✓
Translation defects	*Linux*	FPU+`siginfo`	Cause exception, then handler reads real EIP from saved FPU registers	[22]	✓	✓
		`rdfsbase`	(Privileged) instruction returns FS base value != `prctl(ARCH_GET_FS)`	[22]	✓	✓
		`syscall` EIP	When instruction ends the engine does not write expected EIP to `rcx`	[22]	✓	✓
	Windows	HW counting	Make DR registers point to code and set up SEH, then count exceptions	dbg	✓	✓
		`int 2d` (EAX=0)	Set up SEH, then check if Windows increments Context->EIP by 1	[9]	✓	✓
		`int 2e`	Check EIP value in EDX after system call (32-bit only)	[22]	✓	✓
		PEB WoW64	Check if PEB64 debug-related flags coincide with PEB32 ones	[10]	✓	

it does not trigger a single-step exception when executing JIT-ed code corresponding to such `.text` addresses. DynamoRIO may or may not do that depending on the locations; we reported a bug for it in context handling too.

As for established detection methods, we observe that DynamoRIO mishandles certain usage instances of the `int` instruction on Windows: this happens with `2d` for single-stepping and with `2e` to perform syscalls on 32-bit installations. We contributed to existing discussions on the project page by providing our crashing instances.

DynamoRIO is not affected by the *PEB WoW64* detection described for Pin in [10], which looks for discrepancies between the PEB32 and the PEB64 maintained by 64-bit Windows for 32-bit code to support inspection from 64-bit processes. We speculate that Pin masks the PEB32 `NtGlobalFlag` to be zero as in a native execution but forgets to do so for the PEB64, where we find a `0x70` field value (i.e., heap settings known to reveal debuggers).

3.3 Engine-specific Methods

Table 3 details 12 working methods (1 new) derived from manual analysis of an engine. As new detection we found out that for a non-relocatable ELF program DynamoRIO, similarly to the `gdb` debugger [21], enforces a *heap translation* by placing it within a short distance (<2000 bytes) from the end of the `.bss` section.

However, we believe the most interesting findings came from testing the eXait detections [13] (Pin 2.x, Windows) on newer Pin releases and looking for DynamoRIO counterparts. Some of these techniques read the list of loaded code modules from the PEB of a Windows application. DynamoRIO and newer Pin releases (3.x) load their runtime libraries without recurring to ordinary facilities, hence the modules do not appear in this list and the detections fail. Nonetheless, both allocate memory with special characteristics: we identify memory-mapped image files and inspect the associated file name to find modules. This readily revamps obsoleted detections for:

- *strings* found in a module, such as *"Intel(R) X86 Encoder"* and *"PIN_SetDebugMode: Unknown option."*;
- *runtime exports* found in `pinvm.dll`, such as the `PinCommitHashC` and `PinWinMain` functions;
- *tool exports* present in user-compiled Pin tools, such as the `ClientIntC` function or the `__pin_tls` storage.

These artifacts can be found on Linux too. Locating mapped files then helped us with two more detections. For the *section names* one, which looks for sections typical of `pinvm.dll` or of the user-supplied tool, eXait looks for `.charmve` or `.pinclie` in either module. We found that after the major changes of Pin 3.x `pinvm.dll` features peculiar `.CRT`, `.pinvers`, and `STLPORT_` sections, while user tools show also a `.pinclie` one in addition to those three. Then for the `argv` detection, which looks for copies of command-line arguments made by the engine, eXait first locates the `.pinclie` section of the user tool, then retrieves at 4 bytes (32-bit Pin) from its start a pointer to the `argv` copy. With manual analysis, we found that newer releases place such pointer 4 bytes later.

Table 3. Detection methods stemming from manual analysis of a DBI engine's working. All the methods listed below belong to the *memory consistency* category, except for the first one and the last two as they belong to the *process behavior* category.

	OS	Detection	Description	From
DRio & Pin	*Linux*	Env. variables	Engine introduces environment variables with recognizable names	[22]
	Linux/Win	Strings	Presence of strings in memory that are distinctive of the engine	[13]
		Runtime exports	Engine runtime is a dynamic library with recognizable names for exports	[13]
		Tool exports	User-supplied DBI tool exports functions that have recognizable names	[13]
DRio	*Linux*	Heap translation	Engine places the heap near the end of `.bss` section	dbg
Pin	*Linux/Win*	`argv`	Multiple, unwanted occurrences of `argv` strings in memory	[13]
		Cache signature	Memory regions used by JIT compiler start with magic bytes `0xfeedbeaf`	[35]
		CTX restore	Standard code sequence for restoring CPU registers when leaving user tool	[22]
		Section names	Peculiar section names for DBI runtime and/or user-supplied tool	[13]
	Windows	INTx-JMP-NOP	Distinctive code sequence found in RWX pages (repeated with x=x+1)	[13]
		PinADX PEB	GDB-server debugging interface ("PinADX") alters debug-related PEB flags	[10]
		PinADX pdbgport	NtQueryIP(ProcessDebugPort) returns info=1 when PinADX is active	[10]

As for DynamoRIO, we found strings (e.g., *"Copyright @ DynamoRIO"*), tool exports (`dr_main_client`), and runtime exports (dozens of symbols starting with `dr_`), but no peculiar section names or duplicated arguments in memory. An obstacle on Windows was that DynamoRIO alters `NtQueryVirtualMemory` results to hide regions from `dynamorio.dll`. We located its runtime by blindly probing with `NtReadProcessMemory` every DLL base address valid for Windows ASLR policies, and exploring the PE header for the code module when we found one.

We were unable to revamp two eXait detections. The first method enumerates the handles of the running application and uses `NtQueryObject` (with object information kind `ObjectNameInformation`) to look up objects whose name starts with *"PIN_IPC"*. Recent releases of Pin no longer use conspicuous named objects, nor DynamoRIO does. The second method checks the beginning of each valid memory region for the presence of the addresses of 4 Windows API functions[5]. The eXait Black Hat talk [13] did not detail it: in our understanding, it exposed distinctive DLL import information from the `.idata` section of the `pinvm.dll` runtime. The API selection used for detection is no longer effective already in Pin 2.14; as for Pin 3.x, we observe that the first few imported functions can slightly vary among subsequent releases. On the contrary, those are stable in the `dynamorio.dll` runtime (at least for the releases 7.1-8.x). As DynamoRIO is open source, a remedy to avoid fingerprinting would be to alter the ordering of the imports and/or add unused ones at link time, similarly to what malware authors do to deceive import hashing analyses used in malware classification.

3.4 Other DBI Frameworks

As we mentioned before, Pin and DynamoRIO are not the sole DBI systems available to researchers. Generally speaking, they are the most mature and compatible solutions to date. Nonetheless, in some scenarios users opt also for other systems, for instance to have a more direct control of the instruction instrumentation process. In the following we report (partial) detections for three systems that recently witnessed a good deal of popularity.

Dyninst. The Dyninst framework [7] provides users with primitives to write a mutator program that runs as an independent process and can control and modify a target application using the debugging interface of the OS. Dyninst also comes with off-the-shelf program analyses (e.g., slicing, liveness analysis). In security research, the DBI primitives of Dyninst have been used, among others, for control-flow integrity defenses [24]. Furthermore, it provides static binary rewriting features that are quite popular in some areas, especially software testing.

[5]Those functions are not the same as the ones from the trampoline detection of Table 1. The obsolete detection discussed here looks for `LdrLoadDll`, `LdrGetProcedureAddress`, `ZwSignalAndWaitForSingleObject`, and `ZwClose`. For trampolines, the functions affected by Pin are `KiUserExceptionDispatcher`, `KiUserApcDispatcher`, `KiUserCallbackDispatcher`, and `LdrInitializeThunk`.

As the compatibility of Dyninst with Windows code is presently limited (e.g., x86-64 code is not supported), we tested it for detection attempts on a Linux platform and using an empty DBI mutator program. We could observe increased process statistics for I/O bytes and, to a greater extent, VM peak, albeit the footprint of the runtime seems lower than for instance with Pin. The runtime, visible as a module `libdyninstAPI_RT.so` loaded in the address space of the controlled application, also registers a conspicuous `SIGTRAP` signal handler. We observed the presence of additional regions with RWX permissions and of an environment variable `DYNINSTAPI_RT_LIB`, while the addresses reported in the FPU context detection are different from those the program expects. Also, Dyninst did not pass the module load latency test provided by the authors of [22]. As our confidence with Dyninst is more limited than with Pin and DynamoRIO, we do not rule out the possibility that more detection techniques could become effective once a user enacts specific instrumentation actions from a (non-empty) mutator.

QBDI. The QuarkslaB Dynamic binary Instrumentation (QBDI) system [31] is a modular DBI framework that seeks to support multiple environments. To this end, it builds on the Machine Code representation of the LLVM compiler, which supports all the major architectures and comes with well-tested assembler and disassembler tools. QBDI can currently operate on the following platforms: Linux, macOS, Windows, and Android.

First presented at the end of 2017, QBDI is relatively young compared to other solutions: as of now, the full set of its instrumentation capabilities is available only on the x86/x86-64 architectures, and several compatibility features (e.g., signals, multi-threading, C++ exceptions) are missing. In spite of that, QBDI has seen several applications in the security literature, such as in code deobfuscation [8] and fuzzing [16] research. Also, it is known among practitioners for being of the backends of the popular Frida reverse engineering framework.

On Linux, QBDI can operate via `LD_PRELOAD` injection, or by having its runtime load the target code as a library or as C/C++ code compiled alongside the user tool. On Windows, the injection mode is not supported yet and the compatibility with some constructs is limited[6], which reduced our testing possibilities. The detections that succeeded in either mode and OS were: additional regions (e.g., from `libqbdi_tracer.so` and `libQBDI.so`, respectively, on Linux), egg hunting, available handles, I/O bytes, resident set size, VM peak, module load latency, FPU context, SMC in-block, trap flag, syscall tracing, and the Linux tests of Table 2 for translation defects.

QEMU with User Emulation. The QEMU emulator [4] comes with a mature dynamic binary translator that can convert binary code from different guest architectures to the host one. Historically known as a whole-system emulator, QEMU also offers a User Mode emulation for single Linux processes, which hereafter we refer to as QEMU-User for brevity. QEMU-User executes program code in an emulated CPU, forwarding system calls to the host kernel. Unlike most DBI systems [9], QEMU-User supports cross-architecture execution, provided that any dynamically loaded library the program uses is available on the host as compiled for the guest architecture.

QEMU-User and more generally QEMU were not designed originally with user-provided instrumentation in mind. Researchers, however, have routinely customized its Tiny Code Generator (i.e., the JIT component) to add their probes. For instance, several coverage-guided fuzzing solutions build on QEMU-User to test binaries [16], more recently also in combination with sanitization [15]. The DBI detection sequences listed next may thus be of potential interest as an implementation gap to target in anti-fuzzing techniques [17, 20].

The following detections succeeded in our tests: adjacent pages, W⊕X violation, I/O bytes, VM peak, the latency tests of Table 1, FPU context, signal mask, the (Linux) tests of Table 2 for translation defects, and heap translation. When attempting the `stat` EIP detection, we noticed that QEMU-User exposes the expected address for EIP, but does not emulate the values of several other `/proc/self/stat` fields such as *startcode* and *endcode* (which represent the addresses above and below which program text can run, respectively).

Wrap-up. The many detections that proved to be effective for the three frameworks above back the commonly shared belief that a dedicated adversary can detect the presence of a DBI system by drawing from a plethora of

[6] Among others, QBDI crashed when attempting Windows tests with guard pages and debugging-related instructions (e.g., `int 2d`, `icebp`).

techniques of varying complexity. We believe that more detections may be discovered through manual analysis of the internals of each engine, as for the techniques of Table 3 for Pin and DynamoRIO.

To the best of our knowledge, prior research did not cover detections for QEMU-User and Dyninst, while for QBDI we report many new techniques compared to the work of Kirsch et al. [22]. We leave a more thorough evaluation to future work as the development process of these frameworks continues, in the hope that our research may contribute to ameliorating their transparency and compatibility characteristics.

3.5 Discussion

Recent studies [10, 30] suggest that malware frequently exposes DBI by targeting either its distinctive artifacts or common flaws of dynamic analyses such as induced slowdowns. We noticed similar occurrences with software protectors: for instance, PELock looks for FPU artifacts typical of DBI, while VMProtect breaks DBI with timing attacks and *trap flag* updates. We also saw how previously unreported anti-debug sequences can expose DBI. Security users should be aware of these possibilities, and may decide to extend their tools with logging machinery for likely expected attempts. The lists of detections outlined in the previous sections summarize, actualize, and ultimately advance a decade of reverse engineering work from the security community on DBI transparency.

They are not meant to be exhaustive in an absolute sense though. An adversary may find more implementation gaps by studying the code base of DynamoRIO, while closed-source engines like Pin demand more black-box testing. Another opportunity may be looking for permanent artifacts caused by the use of debugging APIs by the engine, for instance to attach to an already existing process or when Pin on Windows injects a boot routine [34].

In light of the many detections that resulted effective in our investigation, we came to think that sharing them in a publicly available collection could help DBI architects and users to come up with stronger systems.

Lately, researchers have proposed mitigation libraries that run alongside user tools to intercept and deflect detection attempts or inconsistencies [9, 14, 30]. These libraries currently protect 32-bit Windows code running on Pin (and only [9] supports its 3.x releases). The countermeasures in these works are of two kinds: shepherding memory accesses, in order to manually enforce memory consistency, and massaging the output of specific OS queries and sensitive instructions, as for instance with `VirtualQuery`, `fnstenv`, and `rdtsc`.

At the time of writing, a good deal of protection can be found in the BluePill malware analysis system [10], which extends the memory consistency and FPU context mitigations of [9] with countermeasures for attacks targeting: high latency (generalizing the `rdtsc` heuristics of [30]) and trap flag (Table 1), NoDebugInherit, `int 2d`, `int 2e`, and PEB WoW64 (Table 2), and the Windows-specific artifacts of Table 3. Hence, it does not cover the other detections discussed for process behavior (including the one for TLS slots that [14] mitigates instead), time, and translation defects. Also, unlike [30], its memory access shepherding does not hide trampolines (Table 1).

On a different note, our readers may wonder whether any of these detections could apply to whole-system DBI solutions. Oftentimes, these solutions build on and extend the Full-system emulation of QEMU. The differences between single-process and whole-system DBI engines run deep, as their internal workings diverge significantly.

Generally speaking, while address space sharing and contention for OS resources are absent in whole-system approaches, temporal and other behavioral variations, as well as implementation gaps, remain an important source of concern. A realistic emulation of a full architecture and of peripheral devices is a daunting prospect in the presence of adversaries looking for variations in behavior: the problem of detecting system emulators has been the subject of many studies (e.g., [27, 32]), especially in malware research. For the Full-system emulation of QEMU, Zhang et al. detail in [36] a partial list of implementation gaps, such as General Protection exceptions not being raised as in hardware upon, e.g., accessing a reserved MSR register. Recently, Dovgalyuk et al. in [12] report on incorrect FPU context information in QEMU, using the detection that D'Elia et al. [9] wrote for Pin.

4 REFLECTIONS AND OUTLOOK

Following on our architectural considerations and practical evaluation, our readers may be left to wonder whether DBI detection is an impending threat that can undermine the soundness or completeness of their DBI-based analyses. Previous research has not reached an agreement [9, 22, 30]. However, we can make two general remarks.

The first is that researchers when prototyping an idea may not be willing to give up on the flexibility and ease of use of DBI for more transparent emerging technologies. Sometimes a DBI-based functional prototype can be followed by an implementation that is more robust and demanding from an engineering standpoint, building on static binary patching/rewriting (e.g., for software-fault isolation) or hypervisor-assisted (e.g., for execution tracing) instrumentation when this is possible. However, often is the case that such alternatives are not viable: static methods struggle with dynamic code modification and loading, while hypervisor-assisted schemes offer primitives that are too coarse-grained [9]. Recently, EFI [18] appeared as an open source DBI-alike system that uses the `vmfunc` instruction for EPT switching, making it possible to have user analysis code with direct access to the memory of a program and isolation enforced by the hypervisor.

The second concerns the outlook for DBI engines in security. After libdetox [28], the community seems to have no longer worked on systems designed with provisions for such research. Even new systems like QBDI that try to extend the application area of DBI (e.g., to mobile scenarios) do not advance the transparency of the approach. D'Elia et al. [9] observe that newly available technologies like executable-only memory may help in this respect. In Section 3.5 we discussed how current mitigation libraries try to intercept and deflect detection attempts by running alongside user tools. Each countermeasure comes with a different cost. Those related to memory contents can induce high slowdowns [18] as they may have to shepherd memory accesses on behalf of the engine. However, this cost may be acceptable for e.g. fine-grained malware analysis, and design work may help reduce it. On the other hand, deploying heuristics for timing attacks, OS queries, and other defects may be cheaper if they activate only upon probing. At the same time, these should not be too easy to subvert once their details are known to adversaries. We hope our systematization work can foster research in this area.

REFERENCES

[1] Aguila. [n.d.]. NtQuery reverse engineering, coding (blog). https://ntquery.wordpress.com/.

[2] Ange Albertini. [n.d.]. corkami (blog). https://code.google.com/archive/p/corkami/.

[3] Walied Assar. [n.d.]. waleedassar (blog). http://waleedassar.blogspot.com/.

[4] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of the 2005 USENIX Annual Technical Conference (ATC '05)*. USENIX Association.

[5] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM technologies. *Black Hat* (2012).

[6] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent Dynamic Instrumentation. In *Proc. of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM. https://doi.org/10.1145/2151024.2151043

[7] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* 14, 4 (Nov. 2000), 317–329.

[8] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - A Program Synthesis based Approach for Binary Code Deobfuscation. In *Workshop on Binary Analysis Research (BAR)*.

[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 *Proc. of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*. ACM. https://doi.org/10.1145/3321705.3329819

[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. https://doi.org/10.1109/TIFS.2020.2976559

[11] Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro. 2020. Designing Robust API Monitoring Solutions. *arXiv preprint arXiv:2005.00323* (2020).

[12] Pavel Dovgalyuk, Ivan Vasiliev, Natalia Fursova, Denis Dmitriev, Mikhail Abakumov, and Vladimir Makarov. 2020. Non-intrusive Virtual Machine Analysis and Reverse Debugging with SWAT. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*. 196–203. https://doi.org/10.1109/QRS51102.2020.00036

[13] Francisco Falcón and Nahuel Riva. 2012. Dynamic Binary Instrumentation Frameworks: I know you're there spying on me. *Recon* (2012).
[14] Ailton Santos Filho, Ricardo J. Rodríguez, and Eduardo L. Feitosa. 2020. Reducing the Attack Surface of Dynamic Binary Instrumentation Frameworks. In *Developments and Advances in Defense and Security*. Springer Singapore, 3–13.
[15] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. 2020. Fuzzing Binaries for Memory Safety Errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*. 23–30. https://doi.org/10.1109/SecDev45635.2020.00019
[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.
[17] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1931–1947.
[18] J. Hong and X. Ding. 2021. A Novel Dynamic Analysis Infrastructure to Instrument Untrusted Execution Flow Across User-Kernel Spaces. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 402–418. https://doi.org/10.1109/SP40001.2021.00024
[19] Martin Hron and Jakub Jermář. 2014. SafeMachine: Malware Needs Love, Too. *Virus Bulletin* (2014).
[20] Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. 2019. Fuzzification: Anti-Fuzzing Techniques. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1913–1930.
[21] Julian Kirsch. 2018. debugmenot. https://github.com/kirschju/debugmenot.
[22] Julian Kirsch, Zhechko Zhechev, Bruno Bierbaumer, and Thomas Kittel. 2018. PwIN – Pwning Intel piN: Why DBI is Unsuitable for Security Applications *(ESORICS '18)*. Springer International Publishing. https://doi.org/10.1007/978-3-319-99073-6_18
[23] Xiaoning Li and Kang Li. 2014. Defeating the Transparency Features of Dynamic Binary Instrumentation. *BlackHat USA* (2014).
[24] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. tauCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 423–444. https://doi.org/10.1007/978-3-030-00470-5_20
[25] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Association for Computing Machinery, 89–100. https://doi.org/10.1145/1250734.1250746
[26] OpenRCE. [n.d.]. Anti Reverse Engineering Techniques Database. http://www.openrce.org/reference_library/anti_reversing.
[27] 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 *Proc. of the 3rd USENIX Workshop on Offensive Technologies (WOOT'09)*. USENIX Association, USA.
[28] Mathias Payer and Thomas R. Gross. 2011. Fine-Grained User-Space Security through Virtualization. In *Proc. of the 7th ACM SIGPLAN/SIGOPS Int. Conference on Virtual Execution Environments (VEE '11)*. ACM, 157–168. https://doi.org/10.1145/1952682.1952703
[29] Daniel Plohmann and Christopher Kannen. 2012. AntiRE - An Executable Collection of Anti-Reversing Techniques. https://bitbucket.org/fkie_cd_dare/simplifire.antire/src/master/.
[30] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontata, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware *(DIMVA'17)*. Springer International Publishing. https://doi.org/10.1007/978-3-319-60876-1_4
[31] QuarksLab. 2017. QuarkslaB Dynamic binary Instrumentation. https://qbdi.quarkslab.com/.
[32] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting System Emulators. In *Information Security*, Juan A. Garay, Arjen K. Lenstra, Masahiro Mambo, and René Peralta (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–18. https://doi.org/10.1007/978-3-540-75496-1_1
[33] Ricardo J. Rodriguez, Inaki Rodriguez Gaston, and Javier Alonso. 2016. Towards the Detection of Isolation-Aware Malware. *IEEE Latin America Transactions* 14, 2 (2016), 1024–1036. https://doi.org/10.1109/TLA.2016.7437254
[34] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. 2010. Dynamic Program Analysis of Microsoft Windows Applications. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS) (ISPASS '10)*. IEEE, 2–12. https://doi.org/10.1109/ISPASS.2010.5452079
[35] Ke Sun, Xiaoning Li, and Ya Ou. 2016. Break Out of the Truman Show: Active Detection and Escape of DBI. *Black Hat Asia* (2016).
[36] Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang. 2018. Towards Transparent Debugging. *IEEE Transactions on Dependable and Secure Computing* 15, 2 (2018), 321–335. https://doi.org/10.1109/TDSC.2016.2545671