On the Dissection of Evasive Malware

Daniele Cono D'Elia, Emilio Coppa, Federico Palmaro, Lorenzo Cavallaro

***Abstract*—Complex malware samples feature measures to impede automatic and manual analyses, making their investigation cumbersome. While automatic characterization of malware benefits from recently proposed designs for passive monitoring, the subsequent dissection process still sees human analysts struggling with adversarial behaviors, many of which also closely resemble those studied for automatic systems. This gap affects the day-to-day analysis of complex samples and researchers have not yet attempted to bridge it. We make a first step down this road by proposing a design that can reconcile transparency requirements with manipulation capabilities required for dissection.**

Our open-source prototype BluePill (i) offers a customizable execution environment that remains stealthy when analysts intervene to alter instructions and data or run third-party tools, (ii) is extensible to counteract newly encountered anti-analysis measures using insights from the dissection, and (iii) can accommodate program analyses that aid analysts, as we explore for taint analysis. On a set of highly evasive samples BluePill resulted as stealthy as commercial sandboxes while offering new intervention and customization capabilities for dissection.

***Index Terms*—Malware analysis, evasion, dissection, red pill, dynamic binary instrumentation, reverse engineering, sandbox.**

I. Introduction

The interest of security professionals in designing systems and techniques to analyze and characterize malware behavior is at odds with the intention of malware writers, who constantly look for new ways to slip through the cracks of automatic solutions and employ assorted anti-analysis measures to hinder manual dissection. Dynamic techniques in particular extract information from the actions taken in a single execution only. For this reason an armored sample can look for the presence of a dynamic analysis system (such as a sandbox or a debugger) and when one is found it may disguise itself as benign, or try to circumvent or even break such system.

Prominent adversarial techniques for dynamic analysis include: *environment evasion*, to detect the presence of automatic systems and manual tools (e.g., by looking for known artifacts or introduced time overheads); *time stalling strategies*, to make an analysis use up its time budget before any harm is carried out; *anti-reversing techniques*, such as anti-tampering, anti-hooking, and anti-debugging sequences [1].

The common analysis workflow for a complex sample involves a high-level characterization of its behavior using automatic analysis systems sufficiently robust to evasions, such as a state-of-the-art sandbox. For a sample deemed worth investigating a manual in-depth code analysis then follows, so to understand its functional capabilities and structure [2].

Manuscript received August 14, 2019. Revised February 11, 2020. Accepted February 11, 2020.

D.C. D'Elia and E. Coppa are with Sapienza University of Rome. F. Palmaro is with Prisma. L. Cavallaro is with King's College London. This paper has supplementary downloadable material available at http://ieeexplore.ieee.org, provided by the authors. The material includes Appendix §A and §B. Contact delia@diag.uniroma1.it for further questions about this work.

For automatic systems researchers have proposed over time increasingly transparent execution monitoring designs, making them more robust to fingerprinting attempts. Automatic systems nowadays provide valuable indicators for the next steps of the analysis, although their output is occasionally inconclusive. Let alone samples featuring new evasions [3], this may occur also with *targeted* malware that looks for hardware or software characteristics of the specific organization or industry it is destined for [4], and with *trigger-based* malware that stays dormant unless a specific trigger occurs [5].

Adversarial behavior for the subsequent manual stage has on the contrary received less attention from academia. Analysts today still spend a good deal of their time facing detection techniques for their workspaces (e.g., virtualization defects, tools) and the techniques they use (e.g., overheads, debugging artifacts) that resemble those studied for automatic systems. In addition to evasions specific to the manual stage, analysts regularly dismantle techniques that automatic systems used in the first stage already countered or were immune from. As the analysis of complex samples remains a largely manual process, shielding analysts from evasions may bring obvious benefits in the use of their time, provided they can still access fine-grained execution control abilities for dissection.

Contributions: We propose a human-centered dynamic analysis system that can meet the day-to-day workflow of analysts, smoothing the automatic-to-manual transition and enhancing dissection capabilities for the manual stage. Additionally its design favors the interaction between human and machine analyses, an aspect that the state of the art lacks. Our system brings introspection and customization abilities on top of a stealthy execution environment. Its original features are:

- be robust to prominent adversarial techniques for automatic and manual analyses (§IV-A, IV-C, VI-B), and customizable using insights from dissection (§IV-E, VI-B);
- fine-grained execution control capabilities, including a user-space debugger with stealthy live patching (§IV-B), and cloaking of tools popular among analysts (§IV-A);
- let analysts orchestrate program analyses that can aid dissection, as we explore for taint analysis (§IV-D, VI-C);
- be extensible by users in the face of new evasions (§VI-C) or behaviors that deserve close investigation (§VI-B).

To back these capabilities, we choose to pursue transparency by actively hiding run-time artifacts, including those that analysts introduce during dissection such as code patches. In the envisioned design an *observe-check-replace* layer intercepts evasive attempts to hide imperfections, and acts as foundations for upper layers that assist analysts with more high-level capabilities. The design is holistic: it offers an environment for fine-grained introspection (possibly enhanced by program analysis capabilities) that later acts as an automatic stealthy execution system to validate findings from dissection.

We embody our ideas in a prototype implementation BluePill that fares well in stealthiness compared to commercial sandboxing products specialized in evasive malware, making a solid ground where to start dissection from. We build on dynamic binary instrumentation to hook low-level and high-level behavior of the execution, making hooks extensible by the user without incurring semantic gaps in the process.

We report impressions from a preliminary user study where ethical hackers dissected complex samples with BluePill and with other publicly available solutions. We share our code with the community at https://github.com/season-lab/bluepill/.

II. Overview

In the following we examine the transparency and flexibility issues in the different stages of malware analysis that motivate our work. We then illustrate the envisioned design, discussing its use cases and why ease of extensibility is crucial for it.

A. Scenario

The quest for transparency in execution monitoring techniques has driven the initial automatic characterization phase away from the environments that analysts use right after to dissect prominent samples. The automatic stage commonly takes place in one or more *sandbox* systems that observe and record execution facts for a sample in a controlled environment.

While early designs resorted to an in-guest component to *actively* alter the execution of a sample for the sake of monitoring (e.g., via API hooking [6] to intercept interactions with the OS), recent systems operate *passively* from outside virtualized [7] or emulated [8] guests. This choice results in a reduced attack surface for *red pill* sequences that try to detect analysis systems[1]. Out-of-guest designs trade transparency for higher monitoring complexity from the incurred *semantic gap*, which is the problem of interpreting guest memory contents into a high-level semantic state of the running OS [9].

The subsequent manual analysis instead typically takes place on workstations running VMware or VirtualBox images with different software setups and powerful enough to hold multiple *save points* (live snapshots for trial-and-error analysis when debugging). Although custom loaders and drivers may partially cloak VMs [10], operating *inside* the guest implies that analysts may have to manually dismantle stalling strategies, evasions via time measurements, and virtualization red pills (e.g., CPU idiosyncrasies) that the previous stage addressed automatically. This may require a laborious process of instruction and function call interception and input/output patching for each sample. Analysts obviously have to face also techniques specific to dissection, like anti-tampering and anti-debugging sequences that break their workflow.

Some non-academic works (e.g., [11]) have proposed debugging interfaces for virtualization technologies used also in automatic systems, but to the best of our knowledge those have not gained much popularity among analysts so far. Possible reasons may include complexity/limited capabilities in context manipulation due to the semantic gap, lack of interoperability with customary analysis and monitoring tools, and deploy requirements (we shall return to this in §II-C and §VI-B).

The attentive reader would argue that virtualization artifacts leave bare-metal analysis as the only sound approach to date for observing a sample [12] (albeit real-world embodiments may still be fingerprinted [13]). Let us consider also trigger-based or targeted malware: for those an automatic analysis may only reveal suspicious activities at best. Analysts may try different pre-configured VM images, but eventually build an ad-hoc one based on the insights gained from manual dissection. Previous research suggests program analyses like symbolic execution [14], [15] and taint analysis [4] to aid dissection, but their integration in state-of-the-art analysis systems usually conflicts with their transparency requirements.

These considerations motivated us to pursue a holistic approach to the design of a dynamic malware analysis system that could reconcile the transparency requirements for automatic analyses with the levels of flexibility required by human agents when they take over. This can happen after a successful automatic characterization, or even from the start with targeted or highly evasive samples.

B. Approach

We seek for a system that lets users analyze and control instruction and data flows, both first-hand when debugging and with third-party analyses and in-guest monitoring tools, while providing a transparent environment to the code being executed. To advance the state of the art we envision an environment that is:

i) *easy to deploy*: it should integrate well with existing infrastructure (like tools and VMs) for manual analysis;
ii) *interactive*: in contrast to the fixed working of current systems, the user during the dissection can adjust the configuration in use in light of new findings;
iii) *customizable*: the user can (re)define hooks for events such as library and system calls without requiring deep knowledge of neither the system nor OS internals;
iv) *extensible*: to cope with new anti-analysis patterns, it should be intuitive enough for the user to encode countermeasures.

Providing these features, particularly the first three, in a passive design seemed difficult, especially when operating from outside a virtualized analyzed system. We shall return to this in §II-C and §III-B. We thus explored how an active design against adversarial techniques can support such features with transparency in mind. An active approach may let us manipulate the perception of the environment for a running sample, providing it with the illusion it has reached a victim also during its dissection. To this end we propose an *observe-check-replace* paradigm in the design:

- *Observe:* monitor classes of operations performed on the environment possibly to trigger an anti-analysis behavior.

[1]The terms *red pill* and *blue pill* are popular science fiction memes that refer to the truth of reality compared to a machine-generated dream world. Originally used for CPU emulators detection [16], the expression “red pill” in malware research nowadays may apply to general artifact detection techniques.

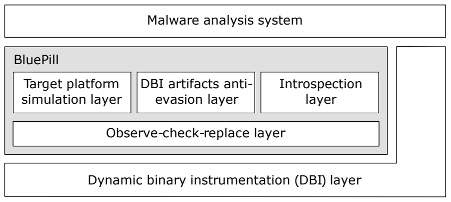

Malware analysis system
BluePill
Target platform simulation layer
DBI artifacts anti-evasion layer
Introspection layer
Observe-check-replace layer
Dynamic binary instrumentation (DBI) layer

Fig. 1. Bird's eye-view of the architecture.

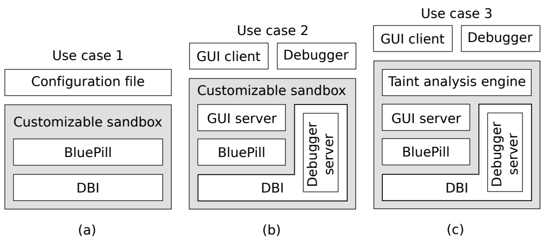


Fig. 2. Use cases for the architecture: customizable sandbox (a), introspection (b), and program analysis (c) scenario.

- *Check:* test whether the outcome differs from what expected in a reference environment chosen by the analyst.
- *Replace:* when it is necessary to fix a divergence, forge the output(s) that would be visible to the sample.

These core primitives are designed to allow the upper layers of the analysis stack to register callbacks that are invoked when a sample executes instances of the monitored operations.

The proposed paradigm supports two categories of operations: *stateless* and *stateful*. For the former case the outcome of an observation is independent of previous observations of that operation. This is for instance the case with the `CheckRemoteDebuggerPresent` Windows API. For the latter case the system needs to track subsequent invocations using an internal state. As an example, `GetAdaptersInfo` queries the OS for network adapters and fails if the buffer argument supplied to hold the result of the API call is too small: on failure it passes back the required buffer size to the caller that in turn uses it to set up another call to retrieve the produced information.

Discussion: Anti-analysis techniques can be more complex than single checks, and rely on an aggregate outcome of combined, possibly parallel execution signatures. For instance, `IsDebuggerPresent` returns whether a process is running in the context of a debugger, and it may seem obvious that it should always return false. However, a sample may then alter the `BeingDebugged` flag of the Process Entry Block (PEB) and repeat the call, which from now on shall return true. Such aspects have to be taken carefully into account in the design of an active monitoring system.

C. Architecture

We now elaborate on how to support observe-check-replace operations and the high-level system features from §II-B.

Modifying the address space of a sample directly (e.g., via DLL injection or from a debugger) would introduce a large number of artifacts and limit the flexibility of the approach. For instance, as modifications would be visible to the sample the system should trap every access to altered regions. The possibility of self-modifying code and anti-disassembly tricks [1] would then require single-step execution to capture low-level red pills, or *direct* system calls that adversaries may make from assembly code instead of using OS user-mode wrappers.

Virtual machine introspection (VMI) techniques offer system-wide analysis capabilities with good transparency. VMI technology builds on reverse engineering work on OS internals to mitigate the semantic gap that occurs when trying to access high-level concepts of the guest [1]. While VMI has led to better designs for passive monitoring systems, the degree of sophistication required to alter aspects of the execution can be a daunting prospect not only for users but also for runtime architects. As an example, while system call interposition is simple to achieve, rewiring a library call to a user-defined function or altering its input/output arguments with newly allocated data structures can be tricky at best. The system also has to treat results provided to a sample and its derived flows (e.g., remote threads, child processes) differently from those for the rest of the system to avoid instabilities and crashes [17].

Dynamic binary instrumentation (DBI) techniques naturally inject extra functionality in a running program [18], which will observe the same addresses (instructions and data) and values (registers and memory) of a native execution [19]. DBI engines abstract away many OS and architectural details to the user, who can write callbacks in high-level languages and access intuitive mechanisms to inspect function inputs and outputs, CPU state, and instructions being executed. Also, they let users invoke external functions (e.g., OS queries, API calls, memory allocations) from the address space of the analyzed program, easing introspection and manipulation activities. DBI thus seems a reasonable choice for the goals we pursue.

We propose a DBI-based design where other layers build on top of the observe-check-replace layer of §II-B. A *target platform simulation layer* controls how monitored sources can reveal information about the execution environment to a sample. Depending on the strain being observed, the analyst should be able to alter specific aspects of the hardware, OS, and software environment to meet its expectations: to this aim the layer provides a tailoring interface to the upper layers. The layer also serves the purpose of hiding the presence of third-party tools used by the analyst in the inspection.

We then devise an *introspection layer* with fine-grained execution monitoring and altering capabilities. The layer exposes debugging capabilities that are a staple for manual dissection. Breakpoints are transparent to the running code, as DBI engines can embed them as part of the trace fetching mechanism [20]. We advance the state of the art with a stealthy live patching mechanism that lets analysts alter a sample's instructions when debugging without having to worry about anti-tampering schemes like checksums. The layer also supports higher-level monitoring at function-call level; as the DBI engine follows code executed in libraries, it does not incur the limitations of binary rewriting or the complexity of VMI.

To conceal artifacts of the underlying execution and instrumentation technology, we complement the two layers with a *DBI artifacts anti-evasion layer*. While DBI meets transparency requirements for benevolent software, researchers have discovered red pills for it (e.g., [21], [22], [23]). We discuss the countermeasures we adopt in §IV-C.

Figure 1 presents the overall architecture. We detail the three layers in §IV, discussing popular ways for a sample to detect analysis systems, how time-based techniques can hinder dynamic analysis, and how our framework copes with such aspects, providing details from our BluePill implementation.

D. Use Cases

As anticipated in §I, our design can back environments that complement and build on each other to assist analysts in multiple steps of their workflow. Figure 2 shows three use cases that we implemented (§IV) and tested (§VI) in BluePill.

a) Customizable sandbox: Instead of being a competitor, this mode can complement state-of-the-art automatic systems. It lets analysts validate findings from manual inspection when dealing with targeted malware or complex evasions: they can devise provisional countermeasures and see if they let a sample reveal its true colors. Analysts may experience a tedious trial-and-error process of building different images only to match shallow checks on software (e.g., Windows MUI packs for Multilingual User Interface) or hardware characteristics (e.g., card readers) that can keep them busy sometimes even beyond a day [24]. BluePill can be instructed to fake results for common observation patterns. This mode also cloaks third-party monitoring tools that are normally banned in automatic systems, with the advantage of not having to reimplement programs that analysts have used for years.

b) Introspection: The goal is to provide an environment where analysts can dissect complex samples flagged by automatic systems or for which the ignition conditions are yet to be determined. We shield analysts from anti-analysis techniques specific to this phase (e.g., debugger red pills, code checksums, detection of popular analysis tools), and from general evasions based on time and red pills for virtualization that would affect also their workstations (§II-A). Compared to current open and commercial systems, the novelty is that the user can (i) inspect and alter execution under the same stealthy features offered for the automatic stage and (ii) tailor environment characteristics on the fly using insights from the dissection.

c) Program analysis: We enhance the previous mode with the ability to define and run complex program analyses over a sample. Previous research hinted at analyses like symbolic execution and taint analysis to ease malware reverse engineering, but their intrusiveness and overheads are a concern for transparency. The design of BluePill can mitigate these issues by allowing analysts to surgically apply them to specific execution portions (§IV-D) and by altering the perception of their overheads (§IV-A). In §VI-C we show uses of taint analysis to dismantle previously unknown anti-analysis measures and to dissect checks in targeted malware.

E. Extensibility

As the intended usage of our system targets human interaction, the role and capabilities of users were pivotal in its design. We envision a mechanism where observe-check-replace hooks can dynamically be adapted based on insights from inspection, resembling other security research [25] where a *human-in-the-loop* paradigm overcomes limitations of machine-based analyses. The user controls the behavior of the system via an initial configuration file, but also dynamically when debugging through a GUI that controls rogue values to be returned for API calls and instructions of interest.

Embodiments of our approach shall not be restricted to providing stealthy ways to explore behaviors only in the presence of known anti-analysis techniques, or they would be helplessly crippled by new evasions. We thus expose DBI instrumentation capabilities to the upper layers, allowing analysts to draw from their experience and current insights from the analysis to tweak existing hooks and, when needed, to add new ones in order to face unsupported techniques or other relevant behaviors.

In §II-C we mentioned the shortcomings of other technologies for implementing such hooks in general. We find DBI APIs sufficiently high-level and simple to be practical also for users with limited experience with the system. Preliminary results seem to back this belief: we present concrete examples of humanly crafted countermeasures for prominent targeted and evasive samples in §VI-C, and report on a different kind of hooks for easing dissection in §VI-B.

III. Comparison to Previous Works

We now review malware analysis research related to the scope of this work, and discuss how our design compares to systems from the literature and solutions used by professionals.

A. State of the Art

Automatic Analysis: As we mentioned in §II-A, solutions based on hardware virtualization and full system emulation have replaced early kernel and user-space monitoring designs for automatic systems. Hardware virtualization enables several forms of system monitoring with little performance overhead, allowing for a quick characterization of many incoming samples [7]. Full system emulation instead lets architects instrument code with custom analyses during the execution, for instance to monitor data flows crossing other components [26]. Both approaches make use of VMI (§II-C) to track the actions of a sample (e.g., invoked system calls) and thus share the benefits and the shortcomings of VMI techniques.

Unfortunately, dedicated adversaries may detect both approaches. Emulators are known for their defects [16] (patching all of them in a practical manner is believed unattainable [27]) and timing differences [28]. Building a transparent VM monitor for hardware-assisted virtualization is infeasible and impractical [29], as an adversary can leverage hardware, resource (e.g., TLB pressure [30]), and timing (e.g., with privileged instructions) discrepancies. [31], [32] investigate execution divergences in different systems looking for environmental differences that a sample could exploit. A normalization step then discards spurious differences for a more accurate comparison between profiles from different sandboxes [33], [34].

Yet malware may evade all available environments, advocating for the construction of a reliable reference system. BareCloud [12] proposes a bare-metal execution platform more robust to fingerprinting as it only monitors network traffic and analyzes disk contents after the execution. Transient effects (e.g., a system call) on the environment cannot be recognized, as any form of in-guest monitoring would violate transparency. The approach is well suited for detecting evasive samples, but less appealing for analyzing their activities.

Manual Dissection: Debuggers are the Swiss Army knife of analysts for carrying out in-depth studies of malware, thus malware writers put significant effort in armoring their samples with techniques to hinder them. Ferrie [35] describes about 80 debugger red pills involving CPU state and instructions, structures like the PEB, Windows system and library calls, and exceptional control flow. More complex detections involve timing attacks and self-checksumming sequences that executable protectors and packers offer as a commodity [36].

Analysts often use cloaking extensions like ScyllaHide and TitanHide based on user and kernel-space hooking, but due to limitations of the approach they cannot hide software breakpoints and code patches, and occasionally break the execution or the debugger itself. The cat-and-mouse game with malware and packer authors also demands for continuous updates, as these tools are regularly detected by recent versions of executable protectors (we report our experience in §IV-C).

A few works dealt with transparent debugger designs. rVMI [11] augments KVM with breakpoints and watchpoints, using the Rekall forensic framework to select processes and navigate kernel structures. It does not deal however with anti-tampering schemes and timing attacks, and is affected by virtualization red pills. MALT [27] uses the x86 System Management Mode to build an enclave where the debugger runs alongside the OS on a bare-metal machine. MALT requires a custom BIOS and communicates with the client via serial port; it uses performance counters to implement single-stepping and breakpoints, and rewrites time and MSR-related instructions to mitigate side effects. Both techniques bring timely enhancements in debugging capabilities for full-system analysis [11] and for rootkits and other ring-0 malware [27]. For a more general day-to-day usage however the technical shortcomings reported in §II-A and §II-C may not be secondary: this view was also reflected by the opinions of the users that we involved in the dissection experiments of §VI-B.

Of a different flavor is the Cobra execution system [37] for dissecting selected code streams in malware. Its runtime exposes primitives to register overlay points: execution goes unhindered until it reaches one, then a localized form of execution takes over. In this setting the runtime acts as an elegant dynamic binary rewriter by breaking code in blocks, invoking analysis routines at their boundaries, and rewriting control transfers and other instructions in a block that can reveal its presence. Unlike other execution technologies available at the time, Cobra could thus withstand popular self-modifying and self-checksumming sequences, and played an important role in the WiLDCAT malware analysis framework [1].

Application of Program Analyses: Program analyses may help in understanding the dynamics of a complex sample. A few works focused on automatic extraction of the target configuration, favoring subsequent in-depth inspections. [38] proposes multi-path exploration to extract a more complete view of a sample when its actions are triggered by specific circumstances (e.g., upon receiving a command from the network or when a certain file is present). [39] uses symbolic execution to identify trigger-based behavior in malware with a main focus on time, keyboard, and network inputs as trigger types. The main shortcomings of both works lie in their limited efficiency and scalability [4], especially for complex samples.

Anti-analysis resistance	BluePill	VT_{emu}	VT_{hw}	Bare-metal	Manual
Environment artifacts	◕	◔	◕	●	◔
Timing attacks	◕	◔	◕	●	◔
Stalling strategies	◕	◕	○	○	○
Targeted checks	◕	◔	○	○	○
Debugger detection	●	●	●	●	◔

TABLE I
ANTI-ANALYSIS RESISTANCE OF CURRENT TECHNOLOGIES.

Capabilities offered to users	BluePill	VT_{emu}	VT_{hw}	Bare-metal
Ring-0 analysis	○	●	●	●
Inter-process analysis	◕	●	●	○
Third-party monitoring	●	○	○	○
Function call interposition	●	◕	◕	○
User-provided analyses	●	◕	◔	○
Invisible breakpoints	●	●	●	●
Invisible patching of sample	●	○	○	○

TABLE II
FEATURES THAT CAN AID ANALYSTS IN MALWARE DISSECTION.

GoldenEye [4] uses speculative execution to address fingerprinting attempts by targeted malware. It dynamically constructs multiple environment spaces during a single execution based on queries to APIs that are labeled beforehand. The execution unit is the basic block: when the end of the block is reached in all alternative environments, it trades space for speed with heuristics that curtail the parallel space, keeping only those settings most likely lead to interesting behaviors.

All these techniques are vulnerable to evasions targeting artifacts of underlying technologies (QEMU for [38], DBI for [39], [4]) and exceptional control flow. To the best of our knowledge, their use is not very common among professionals. In addition to scalability concerns, one reason could be that, as they are fully autonomous systems, they may end up going down a blind alley when dealing with unsupported or unprecedented behaviors. Configuration extraction for complex malware thus remains a compelling open problem.

B. Discussion

Table I summarizes how current approaches are affected by anti-analysis measures. Circles are filled by one, two, or three thirds to indicate when a goal is reached to a small, good, or full extent, respectively. To make a fair comparison, for each technology we depict an *ideal* system that combines features implemented in different works. For emulation-based virtualization we consider PyREBox [40] with the detection of time stalling sequences from [41]. For hardware-assisted virtualization we augment DRAKVUF [7] for Xen with the debugger of rVMI [11] for KVM. For bare-metal solutions we add the capabilities of MALT [27] to BareCloud [12]. For manual analysis we consider a workstation with IDA Pro and ScyllaHide, TitanHide, or Apate [42] for debugger red pills, and the VM Cloak plugin from [43] for virtualization defects.

Bare-metal solutions are vulnerable only to stalling strategies and targeted malware. VMI-based systems using hardware-assisted virtualization fare quite well for classic evasions; emulation-based ones can defuse time-stalling schemes, but can hardly handle targeted samples automatically for the reasons discussed in §III-A. An optimal implementation of our

Name	Technology	Main protections	Dissection capabilities	Integration with other systems	Public	Year
Cobra [37]	DBI	Counter self-modifying and self-checksumming code	Adding instrumentation to a sample	WiLDCAT framework (proprietary)	no	2006
Ether [44]	VMI over VT$_{hw}$ (Xen)	Hide trap flag, patch `rdtsc`, fake MSRs for `sysenter`	Coarse/fine-grained execution tracing, unpacking detection through heuristics	-	yes	2008
MAVMM [45]	VMI over VT$_{hw}$ (custom)	VM monitor with smaller detection surface	Non-interactive analysis data recording	-	yes	2009
BareBox [46]	Bare-metal + kernel agent	*No virtualization artifacts*	Snapshot restore, syscall hooking (driver)	-	no	2011
SPECTRE [47]	Bare-metal (SMM)	*Minimal intrusiveness*	Periodic triggering of analysis modules	-	no	2013
DRAKVUF [7]	VMI over VT$_{hw}$ (Xen)	Mitigate evasions targeting VM monitor	Non-interactive system-wide analysis, syscall hooking	-	yes	2014
MALT [27]	Bare-metal (SMM)	*Minimal intrusiveness*	Debugging, snapshot restore	Remote GDB (left for future work)	no	2015
HOPS [48]	Bare-metal (SMM, PCIe)	*Minimal intrusiveness*	Periodic snapshots, heuristics to locate variables and determine stack traces	Built-in forensics tools	no	2016
LO-PHI [49]	Bare-metal (multi-sensor)	*Minimal intrusiveness*	RAM and disk capture, scriptable analyses (currently post-mortem only)	Built-in forensics tools	yes	2016
Apate [42]	Manual (debugger plugin)	Counter debugger red pills	Debugging	WinDbg	yes	2017
PyREBox [40]	VMI over VT$_{emu}$ (QEMU)	*Inherits from VMI*	Debugging, scriptable low-level callbacks	Built-in forensics tools	yes	2017
rVMI [11]	VMI over VT$_{hw}$ (KVM)	*Inherits from VMI*	Debugging	Remote GDB, built-in forensics tools	yes	2017
VM Cloak [43]	Manual (debugger plugin)	Counter hypervisor red pills	Debugging	WinDbg	yes	2017
Nighthawk [50]	Bare-metal (ME subsystem)	*Minimal trusted code base*	Monitoring RAM/SMRAM for integrity	<does not apply>	no	2019

TABLE III
DYNAMIC ANALYSIS SYSTEMS FROM ACADEMIC AND INDUSTRIAL RESEARCH THAT CAN AID IN MALWARE DISSECTION. PROTECTIONS WHEN REPORTED IN ITALIC COME FROM THE CHOSEN UNDERLYING TECHNOLOGY INSTEAD OF FROM ACTIVE MEASURES TAKEN BY THE SYSTEM.

approach may perform as well as or sometimes even better than existing VMI-based systems. As we detail in §IV-A the platform simulation layer (§II-C) can hide artifacts, control the time behavior of a malware, and expose tailoring primitives for targeted checks. We address DBI-based debugging in §IV-B.

Table II compares analysis capabilities supported by BluePill and current VMI and bare metal-based designs. BluePill falls short for ring-0 analysis as DBI is confined to user space; DBI can follow however system-wide flows like remote threads and child processes. As VMI-based designs operate in the VM monitor, allowing third-party analysis tools to run in the guest to ease dissection is not a possibility (they should be reimplemented outside it, if feasible). This is even more the case with bare-metal approaches where the design itself typically bans in-guest components such as tools [12].

BluePill enables more advanced interposition schemes for system and library calls thanks to DBI (§II-B). Similar considerations can be made for user analyses. While some forms of VMI scripting to register callbacks and reason over events are possible, their introspective power is limited by the underlying forensic framework (while BluePill can make OS queries and API calls directly) and the nature of the operation, as low-level tasks (e.g., controlling instruction sequences) may require modifications to QEMU, and not be practical at all under hardware virtualization. Recently in the bare-metal realm LO-PHI [49] explores analyses of memory and disk snapshots transferred to an external machine, but present performance concerns confine this to happen only upon execution end [49].

Finally, both hypervisor and bare metal-based debugging interfaces offer invisible breakpoints but leave user changes to the code of a sample visible. For hypervisors the page splitting used for invisible breakpoints [51] may in principle be adapted to this end, but the complexity [52] in synchronizing code and data views with arbitrary changes would increase, and this may be one reason behind such present limitation.

Table III summarizes the dissection capabilities and the protections against evasions of several dynamic analysis systems from the literature. We selected systems meeting one or more of the following criteria: the system (a) runs fixed analyses customizable by an expert user, (b) offers shielded interactive capabilities, or (c) minimizes the vulnerable surface. We can see that the features and the goals of BluePill are not fully met by such works or any straightforward combination of them.

For commercial products we can attempt a qualitative comparison. Specialized vendors offer custom virtualization solutions that combine ideas also seen in academic literature (trapping kernel and user-space calls, special instructions, and accesses to areas like the PEB) with robust implementations, supposedly stealth for their proprietary nature. They often ship valuable plugins (e.g., for forensic analysis) and automation infrastructure to encode recurrent actions via scripts. When it comes to aiding dissection, to the best of our knowledge not much is offered, and no solution pursues similar goals or the holistic approach of BluePill. Some systems support mouse/keyboard interaction with the sandbox, leaving to analysts the responsibility not to spook a sample by, e.g., launching a tool. Fewer mention analyses in IDA Pro or Volatily of post-mortem dumps or possibly online during execution, but even in the latter case we found no evidence of technical improvement over the features of open VMI systems (Table II).

IV. FRAMEWORK

Our BluePill implementation targets 32-bit and 64-bit Windows malware. As DBI engine we use Pin [19], which is largely popular in security research as it offers intuitive APIs to place instrumentation at different granularity levels [18]. The manipulations we perform are not specific to Pin[2], thus we believe the approach is portable to other DBI systems.

The section is organized as follows. We first detail the upper layer that controls what the sample sees and asks of the system, including its time behavior. Next we discuss how we provide introspection capabilities for an execution, and how we shield the latter from artifacts that characterize the very same underlying DBI execution mechanism. We then detail how the system presented this far can accommodate program analyses to aid dissection, and conclude by discussing how users can extend BluePill with insights from the analysis.

[2]With the exception of its native debugging interface, which for engines that miss one could be devised following the implementation strategy of [53].

A. Target Platform Simulation Layer

To discuss the implementation of this upper layer we will refer to how we alter the environment perceived by a sample when running in VirtualBox with a Windows image equipped with common, conspicuous analysis tools. Choosing a different hypervisor should not require any major changes, as artifacts known for instance for VMware and KVM can be hidden in the same ways. BluePill can also be used in bare-metal setups. As for analysis tools we consider programs serving different purposes, such as IDA Pro, LordPE, ProcessHacker, Scylla, the SysInternals suite, WireShark, and others[3].

Instead of a *cloaked* VM where custom drivers and loaders hide a subset of known VM artifacts (§II-A) we target a vanilla hypervisor setup, installing also hypervisor guest additions as they provide handy features normally banned in analysis scenarios. The mechanism we use to hide its artifacts can also be used to tailor features of the environment (we detail them in §IV-E) and meet the expectations of targeted samples.

We build on the observation that samples most notably fingerprint the following aspects of an execution environment:

- *Virtualization.* In addition to red pills for instruction-level differences [16], revealing aspects include contents of system firmware tables (SMBIOS strings, ACPI tables), contents and position in memory of specific structures (e.g., Interrupt Descriptor Table, Task State Segment), and I/O ports (e.g., historically used to detect VMware).
- *Hardware characteristics.* Some feature can reveal either a target victim or a virtual machine: CPU model, core count, MAC address family, number of adapters, presence of smart card readers, disk size, serial numbers, etc.
- *Windows installation.* Samples can inspect context information like time zone, language, uptime, install date and so on, especially targeted one. Hypervisors also introduce characteristic registry entries, processes, and drivers.
- *Applications.* The presence of a specific software program might represent a necessary condition to trigger a payload, an adversary that needs to be disarmed (e.g., a firewall or antivirus product), or more simply a sufficient condition for evasion (e.g., an analysis tool).
- *User artifacts.* Fresh Windows installations are suspicious for a sample, which may look into installed applications, navigation history, recently used files, etc.

While using images with a realistic wear-and-tear state [54] is recommended for user artifacts, the other aspects require that portions of an execution be monitored and altered when needed. We place hooks on the following execution items:

- *Special instructions:* `cpuid`, `int`, `rdtsc` and others can reveal hardware features, elapsed time, and debuggers.
- *Library calls:* we monitor APIs that deal with files, registry keys, GUI events, hardware features, drivers, processes, pipes, DLLs, network, mutexes, and time sources.
- *System calls:* samples can use them to achieve (via `Nt` user-mode wrappers exported by Windows in *ntdll.dll* or via direct ASM calls using for instance `sysenter` or `int 2e`) the interactions described above for libraries more covertly.
- *Windows Management Instrumentation:* WMI queries can reveal OS setup, installed applications, and devices.
- *Process environment.* Aspects of the execution environment like the PEB can reveal processor and system information upon inspection (e.g., CPU cores, local debugger).

We refer the reader to the supplementary material (§A) and the source code for their details. For the sake of a more accurate and effective instrumentation, we place probes at the lowest possible level of the software stack: in this way we handle in a single place multiple library functions that in turn invoke the same system call or helper, as well as direct system calls that malware authors use to hinder manual code analysis.

To identify which operations should be monitored, we started from a basic set of instrumentations and gradually extended it by running BluePill against programs designed to fingerprint analysis environments (including, but not limited to, Al-Khaser, Pafish, SEMS, and VMDE), and analyzing a large body of techniques from white papers and resources from ethical hackers (e.g., [55], [56], [35], [57]). The techniques we discovered fall in 8 categories: artifacts when executing in a debugger, file operations, GUI features, hardware fingerprinting, running processes, registry contents, timing differences and time stalling techniques, and WMI queries.

Some system calls belong to multiple categories: for instance, samples can use `NtQuerySystemInformation` to reveal information regarding processes, perform raw firmware queries, and detect system drivers. Moreover, distinct patterns can oftentimes be dealt with using the same machinery: for instance, cloaking certain registry keys or active GUI windows is useful to hide conspicuous aspects of analysis tools (e.g., IDA Pro, ProcMon) as well as of VirtualBox. Overall, we were able to detect and counter more than 100 instances of anti-analysis patterns in our experiments.

Fast Forward Time: The time behavior of a process is another aspect that we attempt to oversee. Windows offers many time sources and timer functions that a sample can use to let automatic analyses time out before carrying any harm, and to hinder debugging sessions if the analyst does not dismantle them manually. Samples may not just perform multiple sleep calls, but also check the time elapsed across them to detect mitigation strategies based on time fast forwarding.

To counter these techniques, patching time-related calls in a sample independently could result in exposing an inconsistent state to it, i.e., the different time sources would not be synced. One may instead intercept time-related operations at hypervisor level, and artificially accelerate the guest's clock. Such a strategy should however distinguish between operations happening because of the sample from those naturally occurring in Windows libraries and internals, as accelerating time indiscriminately could easily lead to system instabilities [17].

We adopt a scheme that manipulates the timer behavior of a process: although it may not be trivial to implement in general [17], the DBI abstraction facilitates it as it confines the effects to the process under analysis.

Our strategy comprises two parts. For each time-stalling call, we fast forward the execution by accumulating the

[3]More can just be added by enumerating their artifacts in the current hooks.

requested time quantity in an internal data structure, while the execution is actually suspended for a short amount of time only (say a few ms). When we observe multiple calls to some delay primitive[4] with the same time quantity or call site of a previous invocation, we accumulate the requested quantity and let the execution continue: this is useful to defeat adversarial *sleep loops*, where a sample achieves long pause intervals by using one or more short sleep operations repeated in cycle(s).

Eventually we use the accumulator to fake results for time queries that a sample may perform, e.g., once timers expire. We observe that samples can use different timing primitives in tandem (e.g., `rdtsc` and `GetTickCount`) across sequences of sleep operations to check for consistency too. We fake the results for each second query by adding to the value produced for the first invocation a quantity made of the time accumulated for any timer operation, plus a value either constant or proportional to the time elapsed in the VM.

This scheme can counter red pills that measure execution time for instructions that take longer to execute in a hypervisor [30], and we use it also to hide the overhead of DBI and of generic analyses built on top of BluePill such as the taint tracking of Figure 2(c). While this strategy proves to be effective on the samples we study, it does not claim to be general, and like most dynamic analyses is ill-suited against time queries operated via an external time source [29] or indirect techniques[5]. Still, we believe it represents a building block towards a more robust strategy, and its design can hopefully be extended (e.g., considering surrounding instructions [58]) to counter new schemes that may appear in the future.

B. Introspection Layer

We now move to detailing the upper layer for introspection. For instruction-level introspection we use PinADX [20] to present debuggers like IDA Pro with a view of the program state as if the transformations and JIT compilation orchestrated by DBI were not present. Compared to traditional debugging solutions, breakpoints are transparent to the code being executed (§II-C, [20]), defeating red pills for software breakpoints and more general checksumming sequences (found, e.g., in recent strains of the ZeuS trojan). Another advantage is that performance is affected only around breakpoints and when single-stepping [20], while full speed execution is ensured for most of the application: this can be valuable in the presence of unpacking sequences and other heavy-duty operations.

We then devise a new mechanism for stealthy patching in user-space debugging that could be useful in domains other than malware analysis. We allow users to make code changes of arbitrary length when debugging: we redo the JIT compilation of the affected instructions adding trampolines that go unnoticed by memory reads from the sample, as DBI makes them point to the original (non-jitted) addresses [59]. The mechanism can thus defeat anti-tampering patterns and shield analysts in the common practice of altering instructions in a sample to force its internal logic. Optionally, we can hide the pages containing the patches with the technique of [18], which shepherds memory accesses to sensitive regions with a shadow page table maintained by the analysis: in this case we raise an exception to simulate unmapped memory. We use it also in another layer (§IV-C) to provide a consistent view of memory by hiding the artifacts (e.g., sections) of the DBI engine. We refer the reader to [18] for implementation details.

Instruction-level introspection and patching are accessible via the popular GDB remote server protocol from IDA Pro and compatible debuggers. As for high-level introspection abilities, we implement a mechanism to track system calls and library functions that are of interest to analysts. We solve symbols and addresses in a library when Pin loads its image, while for system calls we extract from `ntdll.dll` names and ordinals for the current Windows version.

C. DBI Artifacts Anti-Evasion Layer

DBI engines are transparent to benevolent code [59], but as we observed in §II-C a meticulous adversary can reveal them in several subtle ways. This aspect is crucial when designing DBI-based malware analyses. [18] studies DBI evasions and shows how to counter them with a library of mitigations more comprehensive and efficient than prior attempts [60]. We build on this anti-evasion library to get protection against many evasion attacks for Pin including, but not limited to, leaking the real instruction pointer, probing the consistency of memory permissions and contents, and exposing engine internals.

When developing BluePill we contributed to [18] by dismantling a remarkable technique observed in recent releases of VMProtect and in a few samples from the dataset discussed in the supplementary material. To the best of our knowledge, this evasion was new in the DBI detection landscape. When code causes a single-step exception by setting the CPU trap flag to 1 with a `popfd` instruction, Pin triggers an internal exception and crashes. An adversary program can register a handler for this scenario and also check where it is being called: simply passing the exception from Pin to the application would expose Pin. We thus handle the original exception in Pin, and forge a new one at the next instruction: Pin does not intercept it, and when the program's internal handler does, it is fine with it.

We then address two surfaces left uncovered by [18] with countermeasures tailored to the malware domain: namely, we tackle time-based detections and artifacts when debugging.

The fast forwarding mechanism of §IV-A can handle both low-level time attacks on instructions and branches in DBI [23] and attacks found in malware that measure the overhead of a generic dynamic analysis. In early tests our technique proved to be more robust than the one of [60], which traps reads from the `KUSER_SHARED_DATA` kernel structure shared on a user page and rewrites values using an ad-hoc divisor supplied for each sample by the user. Apparently [60] could not keep up with the possibly more complex timing detection patterns from recent malware and executable protectors that we tried, and also accounts for fewer time sources than ours.

[4]We hook user-space functions and instructions for internal time sources, including high-resolution ones. Hardware timers for drivers are out of scope.

[5]While their use in malware is undocumented, low-level indirect measurements (e.g., with a counter thread [61] or by racing in two threads `nop` with virtualization-sensitive `cpuid` [29]) could cripple BluePill and any existing systems. In our context they may draw however the analyst's attention, who can then patch the problematic sequence.

The combination of [18] and our time mitigations makes BluePill withstand currently documented DBI evasions. For instance, a recent work [23] reports on several old and newly discovered weaknesses of Pin: in our tests BluePill overcame all the patterns that can apply to Windows. As for malware in the wild and DBI evasions we can make two observations. Some techniques typically meant for "other" adversaries can inadvertently reveal DBI: this is the case of the `PAGE_GUARD` page protection modifier that malware writers and packers use as anti-unpacking technique, but is also well-known to challenge DBI engines [18]. In our experiments we also found evidence of malware featuring DBI-specific evasions, for instance in samples protected with PELock that attempt to leak the real instruction pointer using FPU instructions.

As for debugging artifacts, this aspect has received rather little attention in the DBI literature. PinADX might struggle with exception handling patterns from the malware realm, failing to pass a caught exception to the sample being debugged. We wrote an extension that instructs Pin to pass the exception to the sample as the analyst detaches the debugger for a moment, and waits for its reattaching before resuming execution in the sample's handler. We then cloak artifacts that appear under PinADX such as the `BeingDebugged` flag in the PEB being set to 1. We test our implementation against over 100 red pills for debuggers from popular works of Ferrie [35], Leitch [57], and Branco et al. [55], [56]. We also observed that while samples armored with recent releases of VMProtect and Themida detect tools like ScyllaHide, the technique behind our debugger currently goes unnoticed.

D. Integration with Program Analyses

We now discuss how the implementation presented this far can back program analysis capabilities over evasive samples. [18] reports that about 95 works in recent years used DBI primitives to back popular security research. Among this wealth of automatic techniques there are program analyses that could be valuable also in manual dissection: consider, e.g., symbolic execution [62], taint analysis [63], or forced multi-path execution [64]. As a case study, we explore how taint analysis can help experts pinpoint and dismantle targeted checks and new evasions as part of a human-centered feedback-loop mechanism, where the analysis can be applied surgically instead of blindly as in automatic approaches.

Taint analysis can determine which computations are affected by predefined input sources [63]. Some works have explored it in the malware domain, for instance to detect flows of user-entered data leaving a browser's scope [65] or to intercept keystrokes meant for another process [8]. The approach we follow is different as in BluePill data is *selectively* tainted *at the analyst's command.*

We use a fork of libdft [66], which offers byte-level tagging granularity and efficiently tracks data flows across general-purpose registers and memory. We updated its code to work with new Pin releases and Windows prototypes and structures.

We let the user choose when to treat as taint source specific (even individual) library/system calls or memory regions (e.g., the PEB); sources can be configured or disabled in the GUI anytime during debugging, thus not only before execution starts. Selectivity helps also when using distinct taint seeds to distinguish sources, as the large encoding space otherwise required to separately account for many sources upfront could result in a large footprint for the shadow memory of the taint engine, degrading performance and possibly undermining the feasibility of the approach. We found 8 seeds to be enough in our experiments, with a space occupancy of 1 byte per address maintained in the shadow memory of libdft.

E. Customizing and Extending the System

Observe-check-replace hooks are a crucial component in our design, and one of our goals is to let users tweak existing hooks or add new ones to meet the expectations of a sample. As mentioned in §IV-A we arrange hooks by 8 artifact categories. To offer a consistent view of the system across complex sequences of checks, hooks within the same category share a list of features that should be masked (e.g., artifacts) or materialized (e.g., additional languages) in queries about the system. We alter the perception of files, registry entries, running processes, IPC objects (e.g., mutexes), loaded libraries and drivers, hardware and firmware strings, Windows settings (e.g., languages), and GUI elements, as well as information accessible via WMI (e.g., BIOS serial, MAC address, CPU temperature and fan statistics, installed firewalls and anti-virus products). Users can tailor such lists in the configuration file or dynamically when debugging using the GUI (Figure 2). Entries are typically strings that can be added and removed[6].

The simplicity of DBI primitives allows for quick prototyping of hooks to counter targeted checks or newly discovered anti-analysis techniques. For system and API calls, the user can write C++ code that is executed when entering or returning from the function, and register it by specifying the name of the function and the arguments needed for inspection or manipulation (we provide some examples in §VI-C). Also as we said hooks can access Windows headers and functions directly to inspect and alter the state. We are currently working on a mechanism to load hooks dynamically via a DLL so that debugging sessions are not interrupted by changes that need recompilation. From a methodological perspective we would like instead to explore the design of a domain specific language for rewriting API results in active monitoring frameworks.

V. Discussion

BluePill shall not be considered a sandbox, nor an automatic system for analyzing evasive malware. Our goal is to bridge the automatic and manual analysis processes: we offer an environment where dissection can (i) happen without incurring the anti-analysis hassle that characterize either or both stages, and (ii) benefit from capabilities missing in previous approaches, e.g., stealthy instruction patching, cloaking of tools, and surgical use of program analyses (§III-B, IV-D). We target a gap between literature and malware analysis practice, for when in the daily practice the automatic analysis of a

[6]Updates are needed when a sample introduces objects mimicking artifacts of an active monitoring system to test its robustness. We encountered such adversarial strategies only with PEB-related debugger red pills.

complex evasive sample (a well-studied problem per se) is not sufficient and human analysts proceed with its dissection (e.g., for new outstanding threats).

In our process-level active monitoring approach (§II-B) DBI lets us devise an execution environment stealthy but also easy to extend. Analysts can tweak it to detect and react to previously unsupported anti-analysis or targeted checks (we provide examples in §VI-C), dodging many low-level details involving the underpinnings of the Windows kernel. Rewriting more behaviors than the ones currently supported by BluePill is possibly confined to an implementation job. The price we pay for this flexibility is that we cannot analyze ring-0 flows.

The usage of DBI as basic building block has the advantage of better interposition capabilities (crucial for achieving the dissection features listed in Table II) and not having to deal with semantic gaps (§II-A). However it introduces an abstraction layer that could be easier to detect with respect to VMI. In §VI we show that the countermeasures we adopt were sufficient in our experiments for BluePill to go unnoticed by prominent armored malware, and that our users could easily extend the system to deal with evasive and targeted samples that commercial vendors struggled with. We will monitor developments in VMI technology, hoping they will allow us to explore a porting (at least partial) of the approach to it.

Our design is vulnerable to unknown techniques. With bare-metal analysis as the only sound way to go [12], automatically characterizing evasive behaviors is a compelling open problem: even designs in principle stealthy like Ether [44] were later evaded using defects of the underlying technologies or other artifacts. Manual dissection often intervenes when automatic analyses are inconclusive: we speculate that evasive behaviors are unlikely to go unnoticed in this stage, and BluePill may provide users with means to face them.

Traps and Pitfalls: In dynamic analysis systems it is crucial to intercept interactions between a sample and the OS. Process tracking-based mechanisms are affected by known pitfalls originating, e.g., from incorrect replicas of OS semantics or race conditions in state inspection [67]. A benefit of DBI is that interposition takes place in the same process of the code under analysis, enabling direct queries to the OS to reason on the execution state. We took into account the recommendations from [67] in the design and implementation of BluePill to minimize the risk of inconsistent executions.

One must also consider that a sample may carry out its flows using multiple processes (e.g., by injecting code to run a remote thread, by loading a DLL as a standalone program with `rundll32`, etc). DBI engines are equipped to deal with them: we use the Child Process API of Pin to follow execution from child/exec-ed processes and remote threads and control it with DBI. Another subtlety lies in analyzing 32-bit samples on 64-bit Windows. The WoW64 subsystem offers an OS compatibility layer across the two architectures, and a 32-bit sample can use it to exercise further anti-analysis techniques. For instance, Windows maintains also a 64-bit PEB that 64-bit processes can query. But as also the 32-bit sample can access it, the system has to cloak it just like the 32-bit PEB.

VI. Evaluation

We now illustrate a preliminary experimental investigation of our system on a set of outstanding samples. We explore how BluePill can aid analysts in dissecting complex armored malware and in dealing with unsupported anti-analysis and targeting techniques. To this end we collect the opinions of a team of 6 ethical hackers trained in binary analysis and reverse engineering, memory forensics, and virtualization technology. While they are currently CS students, they regularly participate in exclusive reverse engineering and hacking competitions such as DEF CON. Being a minimal risk study, we applied for expedited IRB review. We consider this an informal pre-study that, may BluePill gain the interest of analysts, could pave the way to a later extensive in-field study of usable security [68].

A. Preliminary Tests

The version of BluePill employed in the study was tested for robustness against the top-1000 armored samples in the VirusTotal Academic March 2018 dataset (~64K PE32 samples). To identify samples that attempt evasions, we first inspected the dataset using the official Yara rules for anti-debug and anti-VM detection. Since Yara rules encode patterns that are checked statically, we assigned them with different weights to privilege rules that capture definitely evasive patterns over sequences that might see also legitimate uses (e.g., `FindFirstFile`) and result in false positives. For each sample we summed the weights of the matched rules, then we sorted the samples accordingly to pick the top 1000.

In these tests we seek evidence for how malware in the wild can put pressure on BluePill by drawing from a plethora of anti-analysis techniques. As the dataset comes with no ground truth for evasions, we monitor suspect activities involving creation of files, processes, and registry entries, network communication attempts, and uses of the Windows Crypto API. If for a sample little or no activity is detected, we opt to run it in a commercial sandbox and resort to manual dissection in case of discrepancies. While this strategy would be ill-suited to claim general resistance to evasion—which is not a goal of this work—we found it a reasonable compromise to gain confidence in the robustness of the implementation.

We ran the samples in a VirtualBox VM with Windows 7 32-bit, 4 CPU cores, and 3 GB of RAM. The VM image came with applications, documents, and usage history inspired by common sandbox design guidelines [54]. For external communications, we set up an INetSim+Burp server VM to simulate a number of classic services, as organization rules forbid us from giving the samples unrestricted Internet access.

Each sample executed for 10 wall-clock minutes with time fast forwarding enabled: as DBI degrades execution speed, we conservatively allowed for a larger time budget compared to commercial sandboxes (3-5 minutes). An inspection of the logs backed the expectation that evasive checks typically take place in early stages of execution, i.e., before a sample leaves notable effects on the system. When no suspect activities were detected, the prevalent cause was a crash in the Pin engine.

We measured for each sample how many anti-analysis techniques BluePill dismantled: a significant ~92% fraction of

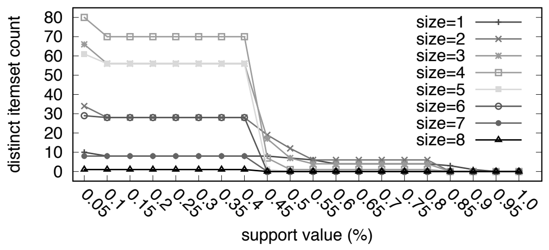


Fig. 3. Number of distinct sets of anti-analysis techniques of a given size for different frequency values across samples.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
DBG				✓	✓				✓	✓		✓	✓	✓		✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓		✓	
FILE				✓						✓							✓	✓		✓	✓		✓	✓	✓		✓	✓								✓				✓	✓	✓		✓	
GUI	✓									✓			✓			✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓		✓		✓	✓		✓	✓			✓		
HW		✓		✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓				✓		✓	✓	✓
PROC		✓		✓	✓		✓	✓	✓			✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓
REG		✓		✓					✓		✓		✓	✓		✓		✓	✓	✓		✓		✓		✓			✓					✓			✓		✓	✓		✓	✓		✓
TIME	✓	✓	✓	✓	✓	✓	✓		✓	✓			✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓					✓	✓	✓
WMI									✓	✓		✓	✓							✓				✓																	✓			✓	

TABLE IV
ANTI-ANALYSIS PATTERNS FROM SAMPLES (BY CATEGORY).

the considered samples resort to at least 4 distinct techniques, while 47% of them adopt at least 8 different detections (we observed as many as 15). Note that the actual number of evasive attempts might be higher: for instance, evasions for other virtualization technologies like VMware, QEMU, or Xen are not tracked by BluePill, but are often present in malware.

To look into the nature of countered evasions, we studied the overlap in used techniques between different samples using itemset analysis. Figure 3 considers for each sample the set of distinct anti-analysis measures it implements, and studies the distribution of the sizes of those sets. In particular, for a given set size a curve shows how the number of distinct sets of that size depends on the minimum frequency of those sets across the samples (*support value*). For instance, there are 80 distinct sets of 4 measures that arise each in at least 5% of the samples (upper-left point in the chart). Similarly, there are 9 distinct sets of 7 measures that arise each in at least 40% of the samples. This shows that assorted combinations of anti-analysis measures are rather frequent in the collection.

As we move to presenting the findings of our informal user study, we remark that further details on the experiments above can be found in §B from the supplementary material.

B. Analysis of Highly Adversarial Samples

With the help of an independent malware analyst[7] we selected 45 samples exercising complex anti-analysis behaviors:

- 15 samples collected by Joe Security in 2013-2019 featuring exotic evasions that their products can handle [69];
- 15 samples selected from the VirusTotal dataset as those exhibiting at least 10 distinct anti-analysis techniques;
- 5 samples shielded by recent versions of the VMProtect, ASProtect, Themida, Enigma, and PELock protectors;
- 10 samples reviewed in 2018-2019 on blogs of firms such as FireEye, Talos, and TrendMicro as particularly noteworthy for hindering automatic and manual analyses.

Using BluePill and monitoring utilities like ProcMon cloaked by it we collected for each sample high-level indicators like dropped or accessed files, contacted entities, manipulated registry entries, and created processes. Our results were consistent with reports collected in Joe Sandbox, a leading commercial solution for evasive malware, and in two cases showed actions missed by it. We used IDA Pro over the GDB interface of BluePill to understand within a debugger the inner structure and capabilities of each sample.

The entire process required ~4 person weeks. We list the categories (§IV-A) for their anti-analysis patterns in Table IV and report their hashes in Table V of the supplementary material. To minimize the risk of bias in the selection process that could favor BluePill, we asked the analyst to pick for the study 12 samples that would best represent the nuisance of adversarial techniques when dissecting complex malware.

We set up a laboratory for our users made of: (A) a Windows 7 VM as in a real-world scenario with common analysis tools for a cloaked VirtualBox, (B) a similar VM with BluePill for vanilla VirtualBox, (C) a Linux system with the QEMU-based PyREBox system, and (D) a Xen-enabled Linux system to run the DRAKVUF automatic analysis platform boosted with pyvmidbg[8]. Due to licensing restrictions and not minor financial aspects we leave out commercial products: yet the reader may refer to §III-B for a qualitative comparison.

Tasks: After a 2-hour tutorial on the systems[9] using sample (42) as a demo, we asked the participants to give the 12 samples a spin in the systems and read the Joe Sandbox reports. Showing them the reports mirrors the workflow of a professional analyst that when about to dissect a sample first runs it in the sandbox(es) available within the organization for an initial characterization (§II-A).

We set up a feedback form and prepared analysis tasks regarding a sample's actions and structure, namely: code revealed/exercised once multiple adversarial techniques are defused; OS interactions and effects on the system that take place using covert techniques and/or in later stages; and facts about protection schemes in place. We defined 3 analysis setups: one with (A), one with (B), and one where users could freely use (C), (D), or both. We made a distribution of assignments such that each user analyzed 6 distinct samples, using every setup overall twice, while each sample was analyzed in each setup exactly once. As the tasks were similarly difficult across samples, no counterbalancing measures seemed to be needed.

Hurdles: What makes the samples bothersome to analyze are the multi-colored ways[10] in which they slow down and break dissection. Adversarial patterns may be laid out in long spread sequences so that analysts may miss some like in (7), be interspersed in lengthy unpacking schemes (35), or even be part of self-rewriting code (40) requiring a laborious patching with breakpoints and single-stepping. Their anti-VM techniques can reveal nearly all hypervisors behind analysts'

[7]With ten years' experience as principal malware analyst in security firms.
[8]Currently the only maintained generic, open VMI debugging interface.
[9]Some participants had used PyREBox (4) and pyvmidbg (3) before.
[10]We report only one sample per technique among all those where present.

workstations (26), while time-based red pills expose debugging activities if not countered (32). Samples may look for many known dissection tools like debuggers and monitoring utilities in covert ways (40), using multiple threads (41) or even altering Windows to prevent their reloading after killing them (39). The VM may be left in an unusable state by wiping essential disk or registry elements (9) and disabling Windows features (36), forcing analysts to use forensic tools over full memory dumps and disks to grasp the effects of a deployed payload. Furthermore they represent a tough proving ground for the DBI artifacts anti-evasion layer of BluePill (§IV-C).

Dissection and Feedbacks: The participants were able to dissect all samples using BluePill, 8 with setup (A) leaving out (26, 35, 39, 40), and 9 with setup (C/D) failing (26, 40, 43). For (A) some adversarial techniques broke the analysis flow; for (C/D) the users exhausted the time budget of 3 hours per sample before completing all tasks.

We received largely positive comments on how BluePill facilitated the dissection process. All the users agreed on how it allowed them to better focus on understanding the actions and the dynamically revealed structure of a sample, without the worry of having to start over as in (A) from the last save point (§II-A) every time an unaccounted or mishandled adversarial behavior kicked in. This was rather evident for samples shielded by packers, especially (35): debugger cloaks like ScyllaHide and TitanHide did not help in (A) when quirky temporal and VM evasions were in place. Some evasions can also be hard to locate in the first place: for instance in (35) are part of lengthy complex unpacking code, (40) is extremely annoying for the analyst as self-modifying code is used to encode them, while in (26) the analyst has to dive into Windows internals.

VMI-based solutions fared better for anti-debugging techniques than (A), but left the users vulnerable to hypervisor detections that they had to counter manually. For (D) users also had to patch checks on CPU cores, as to debug over hardware-assisted virtualization libvmi requires that only one core be exposed to the guest or Windows would incur a blue screen of death. They reported that although debugging via VMI is technically very interesting and powerful, on a first attempt they would still use (A) over (C) or (D) for speed and usability. Criticisms involved difficulties in following and controlling threads alongside the OS scheduler activities, limited assistance in inspecting memory layout of a process and symbols, and coarse grain of the monitoring facilities, if available. The users commended the scripting capabilities of (C) especially if they were to be applied in ring-0 or forensic analysis, but were unhappy with the slow working that adds to QEMU's one, and concerned about the conspicuous custom opcodes used by its in-guest agent for monitoring.

Other positive comments on BluePill involved the interactive altering of the exposed environment (e.g., CPU model and cores) as the expectations of a sample became clear without having to restart dissection, while two users believed the stealth patching mechanism could be useful also in other applications besides malware analysis.

A more interesting fact we witnessed were uses of the system that we did not anticipate. Two participants—of their own accord and without influencing one another—added hooks unrelated to evasions, in order to dig on behaviors surfaced in the Joe reports or to alter their effects. One user observed PowerShell activities in the report of (43) and added a hook that with a regular expression intercepts the launching of scripts via registry entries and APC and yields to the debugger like a semantic breakpoint. The other user was worried about destructive actions from (9) and wrote a hook to rewire calls to components like `bcdedit` (it alters OS boot) to a dummy executable created for the occasion, and another hook to prevent a sample from disabling certain system services and rebooting the machine. The user also liked how these hooks would be reusable when analyzing samples with similar behaviors, e.g., (36).

We came to believe that eliminating anti-analysis techniques creates a baseline to streamline and effectively assist analysts in subsequent core tasks: we look forward to tackling this research direction. In the two users' opinion it would have been significantly more difficult to encode such actions using debugger scripting even in products like IDA Pro, while the VMI mechanisms of PyREBox or DRAKVUF were not as appealing as using DBI primitives, as the capabilities of VMI are currently too low-level in terms of interfaces and also more oriented to observing rather than altering behaviors.

C. Handling Unsupported Techniques

To explore how analysts may turn insights from inspection into extensions to BluePill, we asked the 3 most engaged participants to analyze 3 more samples. The first two are targeted and would stay dormant even on bare-metal systems if the Windows installation does not meet certain characteristics, while the third features an unprecedented assortment of anti-analysis techniques. The users were not given any prior knowledge of the samples. The outcome of this experiment suggests that stealthy introspection capabilities, occasionally combined with taint tracking in the face of lengthy sequences, facilitate valuable insights that may be easy to turn into extensions thanks to the simplicity of DBI mechanisms.

Targeted Malware: The two targeted samples are banking trojans: NukeBot, which attacks French companies, and Retefe, a strain famous for threatening mainly Swiss financial institutions. According to Joe Security, both samples did not initially reveal their behavior in full inside their state-of-the-art sandboxing solution [70], [24]. We briefly report on the findings of the three participants, and how they configured and extended BluePill to analyze the samples in full.

The initial run of NukeBot in sandbox mode (§II-D) revealed a number of files being dropped along with a copy of the Firefox browser, which is then executed resulting in an error message in French. The logs contained an `IsDebuggerPresent` call, multiple checks on free disk space, and a large number of `GetKeyboardLayout` invocations within a DLL loaded after Firefox was launched. When the team moved to debugging it, they discovered that NukeBot uses a DLL side-loading attack affecting old versions of the browser to load a malicious payload embedded in the custom DLL. As the initial stage of the DLL is not obfuscated, they could easily see uses of the `GetThreadUILanguage` and `GetKeyboardLayout` to check user interface and keyboard language against French. As either check is sufficient, the participants opted to add the following hook when returning from `GetKeyboardLayout`:

```
// <windows.h> in W namespace, HKL type for input locale ID
VOID GetKeyboardLayoutHookExit(void* ret) {
  W::HKL *tmp = (W::HKL*)ret;
  *ret = (W::HKL)0x040c040c; /* French */
}
```

A new run in sandbox mode revealed the same extensive credential stealing activities that Joe Security analysts saw only upon updating their VM image with the new layout [70].

For the second case study, when analyzing Retefe in sandbox mode BluePill recorded one `cpuid` occurrence, a number of apparently benign registry queries, an `IsDebuggerPresent` call, and a WMI query, followed by a large number of short sleeps before terminating. The WMI query `SELECT * FROM WIN32_OPERATINGSYSTEM` raised suspicion among the team, who marked its output as tainted and followed taint propagation. While the query returns a data structure with 66 members, Retefe checks the sole `MUILanguages[]` field to see whether `en-US` is the only MUI pack (§II-D) installed in the system.

Joe Security reports [24] that within two days they created a new VM with multiple MUI packs and managed to run Retefe in full. Our users made simple modifications[11] to the existing WMI hook to rewrite the query result: after the changes Retefe dropped the 7zip tool to extract its components from an archive and connect to the TOR network as expected.

New Evasions: Furtim is a sample that in 2016 drew the attention of analysts due to its staggering amount of anti-analysis techniques, a clear sign of the vast expertise of its author [71]. It could evade all known dynamic analysis solutions with the sole exception of the bare metal-based sandbox of Joe Security [3], [71]. Furtim performs over 400 adversarial checks, including registry entries and service executable names from even very rare security programs, and artifacts from major virtualization and sandboxing environments [71].

Furtim terminates prematurely when in a VM or sandbox, and when it detects popular analysis tools it retaliates to the analyst by killing them only at a later stage to disrupt progress. Its checks are rather articulate: for instance it uses `cpuid` to check product brand strings against a blacklist, and later makes a sanity check between CPU model and available cores. When it looks up loaded drivers some entries lead to an immediate termination and others to a later evasion. [3] details checks related to network, DLLs, hardware and BIOS strings, window titles, registry keys, and Direct3D.

When our users analyzed Furtim, BluePill lacked one hook to withstand all the actions listed in [3]: system call `NtEnumerateKey` takes a registry key handle opened via `NtOpenKey` and writes in a buffer the value of the subkey at the given index. Furtim uses it to look for VirtualBox artifacts related to disks, CD-ROM drives, and Direct3D properties. We kept the existence of [3] to us, and for the sake of analysis we informed the team that had all evasions be countered Furtim would drop an executable and add it to autostart programs.

They first ran it in sandbox mode to see what anti-analysis techniques were identified, which included two `cpuid` red pills, a call to `NtQueryInformationProcess` to expose debuggers, and two to `NtQuerySystemInformation` to fingerprint drivers and processes. They thus marked the output of `NtQuerySystemInformation` as tainted and followed taint propagation, which revealed that Furtim uses standard wide-char string processing functions to parse the tainted data. They then hooked such functions to print their arguments and see which strings were processed during the execution. Once suspicious strings containing `VBOX` started to appear, they used our system call tracing feature to intercept possible families of calls that were not already hooked, and inspected the memory pointed to by their arguments to see whether such a string could originate from there. This approach exposed all the uses of `NtEnumerateKey` mentioned above, so they implemented the following code to massage the results by rerouting the query to a random key not present in the system:

```
void NtEnumerateKeyHookEntry(syscall_t *sc,
                  CONTEXT *ctx, SYSCALL_STANDARD std) {
 KEY_INFORMATION_CLASS cl = (KEY_INFORMATION_CLASS)
                            sc->arg2;
 if (cl == KeyBasicInformation) {
  PKEY_BASIC_INFORMATION str = (PKEY_BASIC_INFORMATION)
                               sc->arg3;
  if (wcsstr(str->Name, L"VBOX") != NULL) {
   size_t nameLen = wcslen(str->Name);
   memcpy(str->Name, RANDOM_KEY_WSTR(nameLen), nameLen); }
 }
}
```

Unfortunately Furtim still terminated prematurely. We then suggested to trace library calls with BluePill: by a closer inspection one additional check via `EnumDisplaySettings` not described in [3] emerged. Furtim uses this API to retrieve information on the graphic modes supported by a device, and the participants rewrote the behavior exposed for device `\\.\DISPLAY1` as it revealed VirtualBox. We could not find evidence of this evasion technique in previous literature.

VII. Conclusion

BluePill embodies a holistic approach to reconcile divergent interests and requirements of automatic and manual malware analysis, easing the dissection of complex samples with new capabilities while preserving transparency. As a building block immune from semantic gaps, DBI backs it in useful execution monitoring and altering capabilities, facilitates user extensions and customizations, and paves the way to exploring more program analyses. We share our implementation hoping that security researchers and professionals may benefit from it.

Acknowledgments

We are deeply grateful to Simone Nicchi for his many useful comments. We thank Alberto Marchetti–Spaccamela, Giuseppe Laurenza, Mario Polino, and Pietro Borrello for their help in different stages, our anonymous reviewers for their suggestions, and VirusTotal and their staff for granting us a free academic account. This work is supported in part by a grant of the Italian Presidency of the Council of Ministers.

[11]We report the code in §A from the supplementary material.

REFERENCES

[1] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2008.

[2] D. Plohmann, S. Eschweiler, and E. Gerhards-Padilla, "Patterns of a cooperative malware analysis workflow," in *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, June 2013, pp. 1–18.

[3] SentinelOne, "SFG: Furtim malware analysis," Tech. Rep., 2016, https://www.sentinelone.com/blog/sfg-furtims-parent/ (Accessed: Feb 2020).

[4] Z. Xu, J. Zhang, G. Gu, and Z. Lin, "GoldenEye: Efficiently and effectively unveiling malware's targeted environment," in *Proc. of the 17th Int. Conf. on Research in Attacks, Intrusions and Defenses*, ser. RAID'14. Cham: Springer International Publishing, 2014, pp. 22–45.

[5] D. Andriesse and H. Bos, "Instruction-level steganography for covert trigger-based malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, S. Dietrich, Ed. Cham: Springer International Publishing, 2014, pp. 41–50.

[6] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security Privacy*, vol. 5, no. 2, pp. 32–39, March 2007.

[7] T. K. Lengyel, S. Maresca, B. D. Payne, G. D. Webster, S. Vogl, and A. Kiayias, "Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system," in *Proc. of the 30th Annual Computer Security Applications Conf.*, ser. ACSAC '14. New York, NY, USA: ACM, 2014, pp. 386–395.

[8] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. of the 14th ACM Conf. on Computer and Communications Security*, ser. CCS '07. ACM, 2007, pp. 116–127.

[9] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *Proc. of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11. IEEE Computer Society, 2011, pp. 297–312.

[10] hfiref0x, "Vboxhardenedloader project," 2018, https://github.com/hfiref0x/VBoxHardenedLoader (Accessed: Feb 2020).

[11] J. Pfoh and S. Vogl, "rVMI: A new paradigm for full system analysis," *Black Hat USA*, 2017.

[12] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal analysis-based evasive malware detection," in *Proc. of the 23rd USENIX Security Symposium*, ser. SEC'14. USENIX Association, 2014, pp. 287–301.

[13] A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, and C. Rossow, "Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion," in *Research in Attacks, Intrusions, and Defenses*. Cham: Springer International Publishing, 2016, pp. 165–187.

[14] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting malware analysis with symbolic execution: A case study," in *Cyber Security Cryptography and Machine Learning*, S. Dolev and S. Lodha, Eds. Springer International Publishing, 2017, pp. 171–188.

[15] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Reconstructing C2 servers for remote access trojans with symbolic execution," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, D. Hendler, S. Lodha, and M. Yung, Eds. Cham: Springer International Publishing, 2019, pp. 121–140.

[16] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of red-pills: How to automatically generate procedures to detect cpu emulators," in *Proc. of the 3rd USENIX Conf. on Offensive Technologies*, ser. WOOT'09. USENIX Association, 2009.

[17] F. Besler, C. Willems, and R. Hund, "Countering innovative sandbox evasion techniques used by malware," Tech. Rep., 2017, https://www.first.org/resources/papers/conf2017/Countering-Innovative-Sandbox-Evasion-Techniques-Used-by-Malware.pdf (Accessed: Feb 2020).

[18] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "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*, ser. Asia CCS '19. ACM, 2019, pp. 15–27.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 190–200.

[20] G. Lueck, H. Patil, and C. Pereira, "PinADX: An interface for customizable debugging with dynamic instrumentation," in *Proc. of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 114–123.

[21] F. Falcón and N. Riva, "Dynamic binary instrumentation frameworks: I know you're there spying on me," *Recon*, 2012.

[22] K. Sun, X. Li, and Y. Ou, "Break out of the Truman show: Active detection and escape of dynamic binary instrumentation," *Black Hat Asia*, 2016.

[23] J. Kirsch, Z. Zhechev, B. Bierbaumer, and T. Kittel, "PwIN – Pwning Intel piN: Why DBI is unsuitable for security applications," in *Computer Security*. Springer International Publishing, 2018, pp. 363–382.

[24] Joe Security, "Retefe loaded with new MUILanguage sandbox evasion," Tech. Rep., 2017, https://www.joesecurity.org/blog/7328916856247672770 (Accessed: Feb 2020).

[25] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance," in *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security*, ser. CCS '17, 2017.

[26] Y. Kawakoya, M. Iwamura, E. Shioji, and T. Hariu, "API Chaser: Anti-analysis resistant malware analyzer," in *Research in Attacks, Intrusions, and Defenses*. Springer Berlin Heidelberg, 2013, pp. 123–143.

[27] F. Zhang, K. Leach, A. Stavrou, and H. Wang, "Towards transparent debugging," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 2, pp. 321–335, March 2018.

[28] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proc. of the 10th Int. Conf. on Information Security*, ser. ISC'07. Springer-Verlag, 2007, pp. 1–18.

[29] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, "Compatibility is not transparency: VMM detection myths and realities," in *Proc. of the 11th USENIX Workshop on Hot Topics in Operating Systems*, ser. HOTOS'07. USENIX Association, 2007.

[30] M. Brengel, M. Backes, and C. Rossow, "Detecting hardware-assisted virtualization," in *Proc. of the 13th Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA 2016. Springer-Verlag New York, Inc., 2016, pp. 207–227.

[31] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song, "Emulating emulation-resistant malware," in *Proc. of the 1st ACM Workshop on Virtual Machine Security*, ser. VMSec '09. ACM, 2009.

[32] D. Kirat and G. Vigna, "MalGene: Automatic extraction of malware analysis evasion signature," in *Proc. of the 22Nd ACM SIGSAC Conf. on Computer and Communications Security*, ser. CCS '15. ACM, 2015, pp. 769–780.

[33] N. M. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song, "Differential slicing: Identifying causal execution differences for security applications," in *Proc. of the 2011 IEEE Sym. on Security and Privacy*. IEEE Computer Society, 2011, pp. 347–362.

[34] M. Lindorfer, C. Kolbitsch, and P. Milani Comparetti, "Detecting environment-sensitive malware," in *Proc. of the 14th Int. Conf. on Recent Advances in Intrusion Detection*, ser. RAID'11. Springer-Verlag, 2011, pp. 338–357.

[35] P. Ferrie, "The ultimate anti-debugging reference," Tech. Rep., 2011, http://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf (Accessed: Feb 2020).

[36] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proc. of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. IEEE Computer Society, 2015, pp. 659–673.

[37] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *Proc. of the 2006 IEEE Symposium on Security and Privacy*, ser. SP'06. IEEE Computer Society, 2006, pp. 264–279.

[38] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Proc. of the 2007 IEEE Symposium on Security and Privacy*, ser. SP '07. IEEE Computer Society, 2007, pp. 231–245.

[39] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection: Countering the Largest Security Threat*, W. Lee, C. Wang, and D. Dagon, Eds. Springer US, 2008, pp. 65–88.

[40] Cisco Talos, "Python scriptable reverse engineering sandbox," 2017, https://talosintelligence.com/pyrebox (Accessed: Feb 2020).

[41] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proc. of the 18th ACM Conf. on Computer and Communications Security*, ser. CCS '11. ACM, 2011, pp. 285–296.

[42] H. Shi and J. Mirkovic, "Hiding debuggers from malware with apate," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. ACM, 2017, pp. 1703–1710.

[43] H. Shi, J. Mirkovic, and A. Alwabel, "Handling anti-virtual machine techniques in malicious software," *ACM Trans. Priv. Secur.*, vol. 21, no. 1, pp. 2:1–2:31, 2017.

[44] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proc. of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. ACM, 2008, pp. 51–62.

[45] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "MAVMM: Lightweight and purpose built VMM for malware analysis," in *2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. IEEE Computer Society, 2009, pp. 441–450.

[46] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: Efficient malware analysis on bare-metal," in *Proc. of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. ACM, 2011, pp. 403–412.

[47] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "SPECTRE: A dependable introspection framework via system management mode," in *Proc. of the 2013 43rd Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2013, pp. 1–12.

[48] K. Leach, C. Spensky, W. Weimer, and F. Zhang, "Towards transparent introspection," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, pp. 248–259.

[49] C. Spensky, H. Hu, and K. Leach, "LO-PHI: low-observable physical host instrumentation for malware analysis," in *23rd Annual Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2016.

[50] L. Zhou, J. Xiao, K. Leach, W. Weimer, F. Zhang, and G. Wang, "Nighthawk: Transparent system introspection from ring -3," in *Computer Security – ESORICS 2019*. Springer, 2019, pp. 217–238.

[51] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. ACM, 2013, pp. 289–298.

[52] M. Tarral, "Building a flexible hypervisor-level debugger," *Insomni'Hack*, 2019.

[53] Hex-Rays, "IDA Pin tracer," 2019, https://www.hex-rays.com/products/ida/support/idadoc/1652.shtml (Accessed: Feb 2020).

[54] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis, "Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[55] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies," *Black Hat*, 2012.

[56] G. N. Barbosa and R. R. Branco, "Prevalent characteristics in modern malware," *Black Hat USA*, 2014.

[57] J. Leitch, "Anti-debugging with exceptions," Tech. Rep., 2011, http://www.autosectools.com/anti-debugging-with-exceptions.pdf (Accessed: Feb 2020).

[58] Y. Oyama, "How does malware use rdtsc? A study on operations executed by malware with cpu cycle measurement," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer International Publishing, 2019, pp. 197–218.

[59] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proc. of the 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments*, ser. VEE '12. ACM, 2012.

[60] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, and S. Zanero, "Measuring and defeating anti-instrumentation-equipped malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer Int. Publishing, 2017, pp. 73–96.

[61] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Financial Cryptography and Data Security*. Cham: Springer International Publishing, 2017, pp. 247–267.

[62] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[63] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, ser. SP '10. IEEE Computer Society, 2010, pp. 317–331.

[64] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. of the 23rd USENIX Security Symposium*, ser. SEC'14. USENIX Association, 2014.

[65] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic spyware analysis," in *Proc. of the 2007 USENIX Annual Technical Conf.*, ser. ATC'07. USENIX Association, 2007, pp. 18:1–18:14.

[66] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proc. of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. ACM, 2012, pp. 121–132.

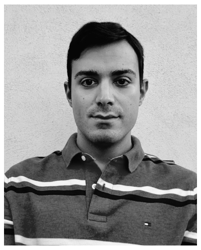
[67] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proc. Network and Distr. Sys. Security Symp.*, 2003.

[68] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 158–177.

[69] Joe Security, "Analysis reports of evasive malware," Tech. Rep., 2018, https://www.joesecurity.org/joe-sandbox-reports-evasive (Accessed: Feb 2020).

[70] ——, "Evasive malware hits French corporations," Tech. Rep., 2018, https://www.joesecurity.org/blog/5668638927855499504 (Accessed: Feb 2020).

[71] EnSilo, "Analyzing Furtim," Tech. Rep., 2016, https://blog.ensilo.com/analyzing-furtim-malware-that-avoids-mass-infection (Accessed: Feb 2020).



Daniele Cono D'Elia obtained his Ph.D. in Engineering in Computer Science in 2016 from Sapienza University of Rome. He is currently a post-doc with Sapienza. His research involves software security and programming language research, with a current focus on malware, code reuse techniques, and code obfuscation.

Emilio Coppa obtained his Ph.D. in Computer Science in 2015 from Sapienza University of Rome. He is currently a post-doc with Sapienza. His research interests include software testing, vulnerability analysis, and reverse engineering techniques.

Federico Palmaro obtained his M.Sc. in Engineering in Computer Science in 2018 from Sapienza University of Rome. He is currently with Prisma researching evasive malware and related dynamic analysis systems.

Lorenzo Cavallaro is currently a Professor and Chair in Cybersecurity with King's College London. His research vision is to develop techniques that automatically protect systems from vulnerabilities and malicious behaviors. He is founder and leader of the Systems Security Research Lab, working at the intersection of program analysis and machine learning for systems security.