

ABSTRACT

Title of thesis: FIGHTING EVASIVE MALWARE
WITH DVASION

Matthew Ryan Gilboy, Master of Science, 2016

Thesis directed by: Professor Rajeev Barua
Department of Electrical and Computer Engineering

Malware is a foundational component of cyber crime that enables an attacker to modify the normal operation of a computer or access sensitive, digital information. Despite the extensive research performed to identify such programs, existing schemes fail to detect *evasive malware*, an increasingly popular class of malware that can alter its behavior at run-time, making it difficult to detect using today's state of the art malware analysis systems. In this thesis, we present DVasion, a comprehensive strategy that exposes such evasive behavior through a multi-execution technique. DVasion successfully detects behavior that would have been missed by traditional, single-execution approaches, while addressing the limitations of previously proposed multi-execution systems. We demonstrate the accuracy of our system through strong parallels with existing work on evasive malware, as well as uncover the hidden behavior within 167 of 1,000 samples.

FIGHTING EVASIVE MALWARE
WITH DVASION

by

Matthew Ryan Gilboy

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Advisory Committee:
Professor Rajeev Barua, Chair/Advisor
Professor Tudor Dumitraş
Professor A. Yavuz Oruç

© Copyright by
Matthew Ryan Gilboy
2016

Acknowledgments

I owe a great amount of gratitude to all who have been so supportive of my graduate studies at the University of Maryland, as well as the research and thesis that has accompanied it. Without this extensive support, this thesis would not have been possible!

I am very thankful to my advisor, Professor Rajeev Barua, for his continued support and for offering me the unique opportunity to join his cybersecurity research team. Professor Barua's expertise with binary rewriting and continual feedback were critical in strengthening DVasion's ability to detect evasive behaviors.

I am grateful for Professor Tudor Dumitraş and Professor A. Yavuz Oruç for agreeing to serve on my defense committee and providing invaluable feedback throughout the year. Furthermore, as my ENEE757 (Network and Distributed Systems Security) instructor, Professor Dumitraş helped me establish a deeper understanding of malicious binaries, while Professor Oruç was an instrumental advisor during my time as a B.S./M.S. student, which eased my transition from undergraduate to graduate studies.

This work would certainly not be possible without the support of Daniel Buetner and the Department of Defense, who were incredibly helpful sources for improving DVasion's ability to detect evasive malware and reviewing this thesis.

The support from SecondWrite LLC's team, including Kapil, Khaled, Aparna, and Tim, has been outstanding throughout the duration of my graduate studies. I appreciate their valuable feedback, especially with helping me understand Intel Pin

and Cuckoo Sandbox in such depth at the start of my research. Furthermore, I am especially grateful to Danny and Julien, not just for providing valuable insight with regards to dynamic binary instrumentation and helping create a sound testing framework, but for putting up with my constant *Star Wars* references this past year. May the force be with both of you!

I am certainly blessed to come from such a supportive family. Thank you Mom, Dad, and Kevin for your love and encouragement throughout my academic studies.

I am especially thankful to Danielle Matteo, whose friendship and loving support has positively shaped me over the last 5 years. My work and thesis are substantially better because of her.

I am lucky to have grown close to so many as a student here at the University of Maryland. Thank you Craig, Brandon, Jason, Jake, David, Mitchell, my friends from Hagerstown 6, and all the rest as it is too many to list. Thank you all!

"Do or do not, there is no try." - Master Yoda

Table of Contents

1	Introduction	1
1.1	Threat Model	4
1.2	Contributions	4
1.3	Outline	5
2	Related Work	6
2.1	Traditional Malware Detection Schemes	6
2.1.1	Static Analysis	6
2.1.2	Dynamic Analysis	7
2.2	Transparent Malware Analysis	8
2.3	Addressing Specific Evasion Strategies	10
2.4	Multiple Analyses	10
2.4.1	Multiple Environments, Single Execution	11
2.4.2	Multiple Executions, Single Environment	11
2.4.3	Additional Work	13
2.5	Summary	13
3	Exploiting Evasive Malware	15
3.1	Structure of Evasive Malware	15
3.2	Proposed System Requirements	18
3.2.1	Dynamic Binary Instrumentation	19
3.2.2	Dynamic Behavior Detection	19
3.2.3	Sandbox Environment	21
4	System Design	22
4.1	Virtual Machine Component	22
4.1.1	Intel Pin	24
4.1.2	Dynamic Taint	26
4.2	Host Machine Component	28
4.2.1	Cuckoo Sandbox	29
4.2.2	KVM	30
4.2.3	Driver	31

5	Experimental Results	35
5.1	Specific Malware Case Studies	36
5.1.1	Dyre	36
5.1.2	Carbanak	38
5.2	Large Malware Dataset Study	38
6	Future Work	44
7	Conclusion	47
	Bibliography	49

Chapter 1: Introduction

Malicious software, commonly known as malware, poses a significant threat to computing resources worldwide as demonstrated by daily attacks against banks, corporations, newspapers, and government agencies. Gen. Keith Alexander, the former director of the National Security Agency (NSA) and former head of the U.S. Cyber Command, cited that U.S. companies lose 114 billion USD annually due to cyber crime [1]; President Obama has stated that “the cyber threat to our nation is one of the most serious economic and national security challenges we face.” [2]. Malware plays a significant role in the execution of such cyber attacks and it must be comprehensively addressed.

Current state-of-the-art enterprise-level malware detection systems are founded on sandbox-based detection, which utilizes isolated environments called *sandboxes* to run suspected malware samples without allowing it to harm the actual host computer [3,4]. Sandboxes are primarily built through emulation or virtualization technology and use a behavior detector that can monitor the program under test to detect suspicious actions, such as sniffing keystrokes, writing to the registry, opening a confidential file, or communicating with a suspicious IP address. An internet simulator may be combined with this to trick malware into believing that it has the

ability to communicate over the Internet, allowing relevant network code to execute that otherwise would have been missed [5].

Despite the research and the industry built around the detection of malicious programs, a largely unaddressed class of malware has emerged leaving many existing sandbox-based detection systems unable to identify the existence of these programs. This malware, known as *evasive malware*, exploits the fact that sandboxes are typically emulated or virtualized environments whose presence may be detectable, malware analysis solutions can only evaluate a sample for a few minutes before moving on to other samples, or the realization that malicious activity could only be performed while running on a targeted system. These malware instances are able to successfully ensure that any malicious code is not executed during its evaluation inside a malware analysis environment, thereby avoiding detection. Evasive malware may employ a variety of strategies to achieve this behavior [6], with common tactics including:

- Waiting for a few minutes to elapse using the system's time or sleep command.
- Waiting for a few minutes to elapse using a look up of time from the Internet; for example, a newspaper's website can likely provide this information.
- Computing useless work for a few minutes before reaching the harmful code.
- Checking for the presence of a sandbox, and not executing harmful code in that case.
- Checking for the presence of human interaction (e.g. mouse movement, key-

board usage), which is usually not found in sandboxed environments.

- Checking for network addresses (e.g. IP address), usernames, files, or system features, and conditionally running the malicious code based on what was found.

According to a recent Lastline report, over 30% of modern malware contain evasive features [7]. Despite its prevalence, existing malware detection schemes are largely unable to successfully identify programs that have this behavior through anti-detection counter measures employed by malware authors, which include code-obfuscation, self-modifying code, and general evasive techniques. A solution that is able to detect this malware in the presence of these qualities is required.

In this paper, we present DVasion (Detect Evasion), a comprehensive solution for addressing all forms of evasive malware. This system exploits the fact that evasive behavior is able to hide through the prevention of malicious code from being executed in a sandboxed environment. DVasion responds to this tactic through a multi-execution approach that forces the execution of these code portions without being prohibitively expensive, primarily because we elect to not maintain program state when evaluating alternate program paths. Our solution utilizes a dynamic binary instrumentation framework to achieve this functionality, as well as an open-source malware analysis system to successfully identify any hidden, malicious behaviors that are uncovered.

1.1 Threat Model

When considering the evasive malware samples that can be exposed through DVasion, the following threat model enumerates the assumptions made by our system. It is important to clarify that these assumptions are based upon the current implementation limitations of DVasion, and not limitations in DVasion’s theory. Chapter 6 explores how these assumptions can be better addressed through future work.

1. The attacker does not try to crash DVasion.
2. Evasive code is hidden by direct branches.
3. Direct, evasive branches are not generated at random locations.

1.2 Contributions

Our work is novel through the depth of results presented and the improved multi-execution approach used to detect evasive malware. Specifically, this paper includes the following enumerated contributions:

1. The exposure of evasive malware utilizing dynamic binary instrumentation, an existing sandbox-based behavior detector, and a multiple-execution approach through DVasion.
2. The development and implementation of a comprehensive solution that is not limited to a specific variant of evasive malware.

3. Results that indicate that the maintenance of program state is not critical for discovering evasive behavior.
4. The usage of static and dynamic optimizations that refine the multiple-execution approach and allow for a satisfactory run-time.

1.3 Outline

In Chapter 2, an extensive overview of malware detection solutions, with a focus on evasive malware strategies, is presented. In Chapter 3, the structure of evasive code within malware is exposed and fundamental concepts that are utilized in DVasion are discussed. In Chapter 4, the architecture and implementation details of DVasion are examined, with experimental results on an extensive set of malware samples provided in Chapter 5. In Section 6, our envisioned future work is detailed, and Chapter 7 completes the work with our conclusions made.

Chapter 2: Related Work

DVasion seeks to establish itself as an effective solution for the detection of evasive malware behavior. This section specifically elaborates on the related work performed in this field of study, allowing for the identification of the strategies and limitations within previous proposals to defend against evasive malware. First, traditional static and dynamic malware detection schemes are discussed, followed by more advanced systems that address sandbox transparency and specific variants of evasive malware. Finally, it addresses the related work that is most similar to DVasion, which perform multiple executions to obtain a greater vision of the true behavior of the analyzed binaries.

2.1 Traditional Malware Detection Schemes

2.1.1 Static Analysis

Much of the early malware detection schemes employed static analysis techniques, which involves inspecting program binaries without running them to recognize if they are malicious or not. This category includes the well-known signature-based detection scheme [8], which compares unknown binary contents to a database

of known malicious samples for a match.

SAFE [9] addresses some of the code obfuscation techniques that thwart static analysis solutions, including complex control flow changes and register reassignments. Their system involves the use of abstracting the code of known malicious patterns and the executable under test, which allows for a direct comparison between the two to identify if any known malicious behavior exists in the binary. In [10], a similar system was developed using the semantics of instructions to detect the malicious behavior within binaries. The work in [11] also involved the use of abstraction to detect the presence of a rootkit at load time, even in the presence of minor changes to the binary image.

These systems are fundamentally limited in a variety of ways, including the use of advanced code obfuscation techniques and self-modifying code. Through these limitations, the code that is analyzed statically may not be the code that is executed, leading to missed, potentially malicious, behavior.

2.1.2 Dynamic Analysis

Seeking to overcome the limitations of static analysis schemes, dynamic analysis involves the execution of a binary and observing its run-time properties to identify malicious samples; such behavioral information may be captured within a sandbox, the protected, isolated run-time environment discussed previously, which can relay the observed behaviors to the host system and indicate if the binary is safe to run without such protections in place. Such environments include CWSandbox [12],

TTAnalyze [13], and Cobra [14], which yield a diverse set of details about the executable gathered at run-time including created processes, Windows API functions called (along with the arguments used for each), and any opened network connections.

Siren [15] extends the abilities of the virtualized environment used for testing by allowing activity to be injected into the system that will generate a known series of network requests. This strategy seeks to identify malicious binaries that involve the transmission of information over the Internet, as such traffic will be discovered after removing what corresponds to the previously injected activity. Panorama [16] observes the correspondence between information access and processing behaviors, a known quality of privacy-breeching malware. Through dynamic taint analysis, explored in Section 4.1.2, it is able to formulate applicable data relationships that may indicate the presence of this variant of malware.

Ultimately, these systems are limited through the observation of a single execution of a program that may not provide a complete analysis of the binary through the use of evasive code techniques. A solution that is fully functional in the presence of evasive behavior, along with code obfuscation and self-modifying code, is a necessity.

2.2 Transparent Malware Analysis

Research has actively explored the development of systems that are able to detect evasive malware by making themselves transparent to the malicious executable,

the hypothesis being that if the system goes undetected, any evasive, malicious code will execute.

Ether [17] utilizes hardware virtualization extensions (e.g. Intel VT) to make itself transparent, which removes the requirement of in-guest software components that are vulnerable to detection. Despite its efforts, self-cited flaws in Intel's VT have left the system vulnerable to detection. While our proposed technique may be detected as well, dynamic binary instrumentation can force the associated condition checks to evaluate differently. Lastline [18] has taken a different approach through full system emulation, which allows for an actual operating system to run on top of an emulator. Emulation makes it difficult to detect the analysis system, but these systems can be affected by the semantic gap between the observed CPU instructions and higher-level concepts like files, processes, and network traffic. DVasion does not have this difficulty, as it utilizes an injectable DLL to obtain this higher-level view. Concerns over host-level detection inspired a disk-level malware detection strategy [19] that is placed in the disk drive. This system utilizes the sequence of disk requests made by a program to detect malware.

Fundamentally these systems are incomplete evasive malware solutions, as they requires the evasive, hidden behavior to execute when their virtualized, emulated, or hidden detection systems are not discovered; in this case, the malicious actions performed would caused the malware be successfully detected. It very well may be the case that malware employs both anti-virtualization/anti-emulation and additional environmental checks, which may cause this behavior to remain hidden given the current date or the host machine's IP address. As indicated by the previous

examples, it is not easy to obtain a transparent system.

2.3 Addressing Specific Evasion Strategies

Some systems have been strategically built to address specific classes of evasion strategies. The detectability of static honeypot tokens and how randomized honeypot token generation is more effective in tricking spyware has been demonstrated in [20], while [21] focuses on detecting malware timebombs that involve the invocation of malicious code at some defined time in the future. The work in [22] addresses execution-stalling malicious code that delays analysis systems to such a degree that the binary's malicious code is not reached, leveraging the fact that a binary can only be inspected for a finite amount of time in the sandbox environment. The critical weakness of these systems is that they are geared toward a specific evasive technique and not a general approach. Furthermore, these systems are not perfect within their own realm. For example, the integration of a cryptocounter within a malware instance can prevent [21] from observing the value a malware is counting down to. Our proposed system is designed to comprehensively tackle evasive malware without any limitations imposed by the type of evasions performed.

2.4 Multiple Analyses

Recently, a new generation of evasive malware detection systems has emerged involving the execution of the binary multiple times, each with different run-time constraints or settings. These systems seek to obtain a more complete understanding

of a binary by reaching new code that would not have been executed in a traditional single-execution system due to hidden, malicious behavior; these systems are most similar to ours.

2.4.1 Multiple Environments, Single Execution

By embracing the idea of running a program multiple times across different environments, the work from Balzarotti et al. [23] and BareCloud [24] compares the observed behaviors on an array of different systems, including a bare-metal reference system similar in nature to prior work [25,26], with the assumption that deviations in behavior are evidence that the program tried to evade a portion of the analysis systems. In addition, MalGene [27] furthers this work through its automation of evasion signature extraction from the obtained results, which can be used to quickly interpret the evasions present. The primary flaw of such systems is that the program under test can still evade all analysis systems through the incorporation of elements like stalling code. When this occurs, there will be no observed deviation of dynamic program behavior and the evasive malware would be incorrectly classified as benign. DVasion is not affected by these weaknesses through a combination of multiple executions and binary rewriting.

2.4.2 Multiple Executions, Single Environment

Two bug-finding tools, Dart [28] and EXE [29], utilize multiple-execution analysis to identify input cases that cause program crashes. These systems are differ-

entiated from the proposed work as they require the source code of the binary and seek to find program bugs. Given that malware is always in an executable form, we will never have access to the original source code.

Moser et al. [30] and Brumley et al. [31] have both proposed systems similar to DVasion, which allow for the execution of multiple code paths within a program in order to find evasive behavior, the key difference being that these systems elect to maintain state in the presence of modifying a binary's control flow. Both of these papers contain limitations that prevent them from a reasonable application to modern malware. Through maintaining program state, these systems require that they are able to modify the contents of the program's register values or address space to correspond with an alternate branch outcome or else the corresponding additional program path will not be executed. Despite improvements made in the latter work, this strategy is not perfect and a malware author could utilize complicated data relationships to hide their evasive branch from future inspection. In this situation, DVasion can still force the program execution in a different direction, as it is not reliant on the ability to maintain program state. Furthermore, the design decisions made in the first paper prevent the system from being applied to multi-threaded applications, as well as those that use signals. The second paper is burdened from a high run-time overhead, with analyses running shy of 30 minutes. DVasion is not affected by these incompatibilities and significantly high overheads as it does not maintain state, allowing it to be freely applied to programs with an overhead that supports a high throughput of analyses.

Limbo [32] recognizes the slowdown applied by strictly maintaining program

state, but it was only designed to detect kernel rootkits. Furthermore, the system restores only the state of the CPU and stack when back-tracing to a previous basic block for further exploration, which fails to undo all previous actions including the creation of a file on the file system. DVasion may be applied to any malware and utilizes clean virtual machine snapshots to ensure that there is no external side-effects to cause a program to act a different way in the additional executions performed.

2.4.3 Additional Work

While no results accompany these related works, we found it important to identify two patent submissions utilizing multiple executions to obtain a greater understanding of a program, with the second patent focused on detecting evasive malware [33,34].

2.5 Summary

Despite the continual and detailed research performed to develop a system that can address the rising class of evasive malware, there has yet to be a fully robust solution proposed that can tackle this issue. Furthermore, the results within the related work involving evasive malware indicate a striking trend: the discovery of new, potentially evasive behavior has never been automatically reported on a large dataset. Previous work [30] has utilized single, fine-grained case study examples to demonstrate new behaviors found; with larger datasets, this work has cited code

coverage metrics to assess their ability to explore a binary further. We strongly believe that code coverage is an inadequate metric to measure our system's effectiveness, as it does not firmly conclude whether or not hidden, evasive behavior was found within the tested programs. To the best of our knowledge, through DVasion's automatic reporting of new behavior that directly corresponds to a modified execution, we are the first paper to present such conclusive results of new behavior within a large, general malware dataset.

Chapter 3: Exploiting Evasive Malware

DVasion is a multiple-execution analysis solution that yields a more complete picture of a binary through the execution of many alternate program paths. Going beyond similar multiple-execution proposals made to address evasive malware, DVasion does not share some of the key limitations identified in Section 2.4. Furthermore, our proposed system includes a component that automatically detects a diverse range of new, potentially malicious behaviors that are discovered and directly correspond to one of the newly explored paths. This section first discusses the structure of evasive malware, which leads to the enumeration of a requirement list for a system that may exploit this code to yield any hidden behavior. Through this, three key system components utilized by DVasion are discussed: dynamic binary instrumentation, dynamic behavior detection, and sandbox-based testing.

3.1 Structure of Evasive Malware

The ultimate goal of evasive malware is to avoid detection within an analysis environment, while maintaining the ability to perform its malicious actions on a legitimate user's machine. As discussed previously, evasive malware may employ a variety of tactics, including checking for the presence of a sandbox or computing

useless work at the beginning of the program's execution.

Irrespective of the evasive strategy employed by the malware, evasion can be accurately visualized at their fundamental unit: an assembly branch instruction. The branch instruction empowers a program to follow one of potentially many paths based on the evaluation of a condition. With DVasion, we focus our analysis specifically on direct branches, which result in a program's control flow to travel to either the branch's target or fallthrough address, which are the addresses specified in the instruction or is the next instruction after the branch, respectively. With two possible execution paths, direct branches are a natural way to represent an evasive condition check that must be satisfied before a malware's malicious behavior is executed. Chapter 6 reflects on indirect branches, which may allow for more than two possible execution paths, and how they will be incorporated within our future work.

To best visualize a direct branch, consider the code snippet below that checks the number of processor cores that are on the host system. This can be a successful tactic in detecting that it is within a virtualized or emulated malware analysis sandbox as these systems are commonly allocated a single core and a small amount of RAM so that many instances of these environments can exist on the same machine, which allows for the simultaneous evaluation of different unknown binaries. The fact that these hardware allocations differentiate significantly from today's commonplace multi-core computers that include multiple gigabytes of memory makes this an effective evasion strategy.

```
1: mov eax, dword ptr fs:[0x30]
```

```
2: test ebx, ebx
3: mov dword ptr [esp+0xc], eax
4: pop eax
5: mov eax, dword ptr [esp+0x8]
6: cmp dword ptr [eax+0x64], 0x2
7: jb 0x403923
8: call 0x4037e0
```

This code has been extracted from an instance of Dyre, further discussed in Section 5.1.1, which involves a direct branch that is used to hide the majority of its functionality. In Line 1, the malware obtains the pointer to the Process Environment Block (PEB), which is located at an offset from the Thread Information Block (TIB); both of these are Windows data structures that contain information about the current process or thread. This value is placed on the stack on Line 3, which is ultimately placed back in `eax` in Line 5. Line 6 is the evasive check within this code snippet, as it results in a comparison of the value `0x2` and the contents at an offset of the pointer to the PEB, which happens to store the number of cores present in the system. Through this, Dyre is able to hide its evasive behavior behind the fallthrough address, given that single-core test environments will cause Line 6 to evaluate to be true and the next instruction executed will be at the branch's target address.

Throughout the remainder of this paper, such target or fallthrough addresses that will yield a new segment of code that has not been previously executed will be

referred to as potential *Evasive Entry Points*, or EEPs. It is clear that the identification of these code segment entry points is critical to finding new paths within a program to explore. In turn, the discovery of new behavior must be accounted for while these paths are explored.

3.2 Proposed System Requirements

As it has been illustrated in the prior analysis of the Dyre code segment, a multi-component system to address evasive malware is required to be successful; ultimately, such a system may be considered in three key parts:

1. **EEP discovery and evaluation.** The identification of EEPs and the ability to force a program's execution to follow the corresponding alternate path.
2. **Behavior detection.** The detection of new, potentially malicious, behavior that can be directly correlated to the execution of a branch's alternate path.
3. **Isolated environment.** The use of an isolated, protected environment to evaluate any suspected malware instances so that no modifications are made to the underlying host system.

The first of these fundamentals may be realized through dynamic binary instrumentation, while the second is provided by monitoring the binary at run-time. The third component may be safely achieved using traditional emulated or virtualized sandbox environments. Each of these are explored in the following subsections,

which allows for an easy segue towards the development and understanding of DVa-sion’s architecture.

3.2.1 Dynamic Binary Instrumentation

We require the ability to discover EEPs and modify a program’s execution path to follow the alternate outcome of a branch when desired, each of which can be achieved through *dynamic binary instrumentation*, or DBI. Primarily used to monitor a program’s execution, DBI frameworks can allow for the observation of events at the instruction level and the modification of a binary’s run-time state while executing. At the instruction level, DBI code may be tuned to detect instructions of a certain opcode, which is essential to discovering EEPs that may correspond to branch instructions. Run-time state modifications can include modifying the thread’s register values, including the instruction pointer register `eip` when considering the x86 architecture; the ability to modify this register can allow for the direction of a program’s control flow to be altered, as this register corresponds to the thread’s current instruction. DBI frameworks are especially helpful as they do not require the program’s source code to run, unlike some previous work cited in Chapter 2. A variety of DBI systems exist, including Intel Pin [35] and DynamoRIO [36].

3.2.2 Dynamic Behavior Detection

We also require the ability to identify new behavior indicative of malicious activity that corresponds directly to an alternate branch outcome. Relaxing this

requirement slightly, we first require the ability to detect the dynamic behavior of the binary in any part of its execution.

Following many of the strategies employed by the dynamic analysis systems discussed in Section 2.1.2, this requirement may be satisfied by utilizing a tool that observes a variety of run-time actions a program can perform. Together, these actions may yield the identification of significant, potentially malicious, behavior. It is important that the DVasion component capturing this dynamic behavior covers a diverse amount of sources, which may contain, but is not limited to:

- Windows API calls made
- Modified Windows registry keys
- Files created, accessed, or modified
- Network traffic
- Processes created
- Dynamically generated or modified code

Information from any of the above input sources may be independently evaluated or combined to yield *signatures* that indicate the presence of a certain activity or program quality; for example, a keylogger can be captured through observing a call to the `SetWindowsHookEx` function when used with a select grouping of function arguments. Existing systems that observe such behavior include Cuckoo Sandbox [3] and Anubis [4].

3.2.3 Sandbox Environment

Lastly, we require an isolated, protected environment to run potential malware samples in, which may use emulation or virtualization technology to avoid allowing the programs to modify the host system.

The ability to capture *snapshots*, or saved states within a sandbox at a moment in time, and revert to them in an effective manner is an absolute must, as we will use this to undo any changes a program is able to make within its sandbox. Without this restore, programs' side-effects will persist across the evaluation of multiple samples, which can build up and affect how additional binaries behave. Previous work in Section 2.4.2 did not perform this complete restore as they simply modify the process's state to evaluate a different code path; this incomplete restore can be problematic with interactions that create external side effects, including the use of files and network activity.

Given that the sandbox will be reverted to a previous, clean snapshot before the execution of different code paths, these evaluations may be thought of as individual tasks that are independent of one another. This realization can be exploited by running these tasks in parallel, given the availability of multiple sandboxes to be used for testing. Therefore, the sandbox of choice must effectively support running multiple instances of it simultaneously.

Many emulated and virtualized environments can provide a sandbox environment that is suitable for protecting the host system, including Kernel-based Virtual Machine (KVM) and VirtualBox.

Chapter 4: System Design

To satisfy the system requirements that were established in Section 3.2, DVa-sion’s architecture incorporates a virtual machine and host machine component, where each plays a key role in the functionality of the proposed system. The virtual machine component is responsible for the instrumentation of the unknown binary, which includes logging the code that has been executed, gathering evasive entry points to be further evaluated, and performing modifications to the executable’s run-time state to allow for these paths to be explored. The host component acts as the supervisor to an array of sandbox environments, each of which contain the virtual machine component and have been instructed to perform a certain task; such tasks include the exploration for evasive entry points or to forcibly cause a program’s execution to follow a certain path. Both of these components are described within this section and may be visualized in Figures 4.1 and 4.2 for the virtual machine and host component, respectively.

4.1 Virtual Machine Component

The virtual machine component is responsible for performing several tasks, which have been enumerated below; we seek to utilize dynamic binary instrumenta-

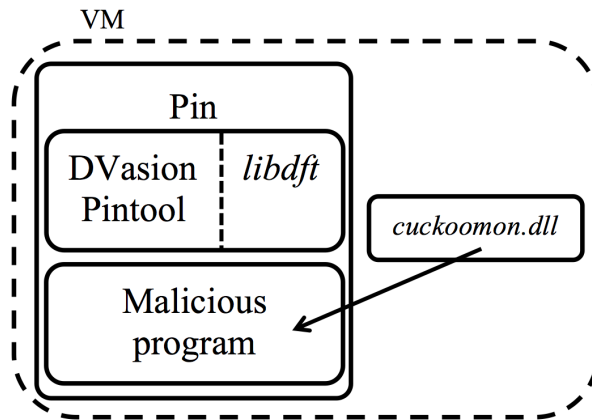


Figure 4.1: An overview of DVasion’s virtual machine component.

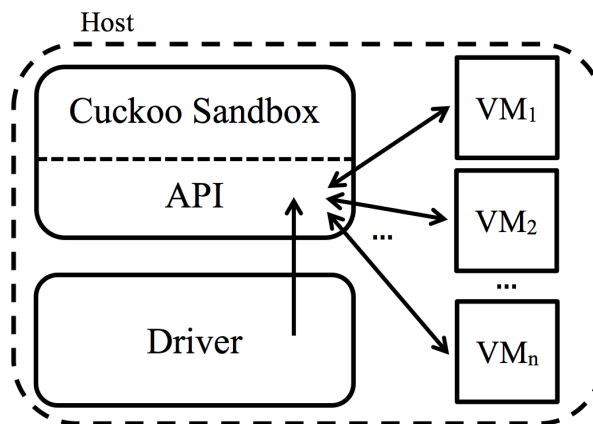


Figure 4.2: An overview of DVasion’s host component.

tion to achieve these functions, so this establishes a requirement list that our choice of DBI framework must satisfy.

1. **Branch discovery.** The dynamic discovery of direct branches whose alternate path (i.e. fallthrough or target address) has yet to be explored.
2. **Visited instructions.** The logging of where the program's execution has traveled.
3. **Run-time state modification.** The manipulation of the program's run-time state, specifically the stored instruction pointer, to effectively change the program's execution path at a given instant.

4.1.1 Intel Pin

Our proposed solution uses Pin [35], a DBI framework developed by Intel that uses just-in-time (JIT) recompilation to instrument a program at run-time. While Pin is closed-source, it provides a rich API that may be used to instrument a binary; such programs are called *pintools* and are written in C/C++, which are compiled as DLLs that accompany the execution of the Pin executable. Pintools primarily consist of two types of routines that are provided to Pin via its API for instrumentation, called *instrumentation* and *analysis* routines. Instrumentation routines are called once when code is about to execute for the first time, and are accompanied with static information like instruction opcodes and direct memory addresses. As mentioned before, these functions have the ability to insert analysis routines that may be associated with certain code from the original binary, and they are called

every time this original code is executed as well. Given the availability of both through the DBI framework, it is clear that any function that requires statically observed information of a program binary can utilize instrumentation routines; additional information that requires the instruction to be evaluated at run-time, like an indirect memory address, may use analysis routines.

As with any use of DBI, the run-time overhead of such analyses must be carefully controlled given that inefficiencies can cause havoc through high instrumentation overheads. Pin runs at a base overhead of 1.17x [37], which is further increased through the use of instrumentation and analysis routines. Several strategies may be employed to reduce this overhead, many of which involve analysis routines. These functions are dynamically called with respect to the program’s execution. Therefore, it is important that such functions are either reduced in complexity, size, or the frequency that they are called. If any analysis computations can be performed statically, these should be placed in the corresponding instrumentation routines. This simplification of analysis routines aids Pin in its effective optimization to inline these function during the recompilation process.

Through Pin, we are able to observe which instructions a binary has executed through instrumenting it; specifically, we monitor the execution of instructions at the basic block level, as this reduces the overhead introduced by the pintool compared to instrumented on a per-instruction basis while preserving a strong accuracy with what code executed. With Pin, basic blocks are single-entrance, single-exit sequences of instructions, which allows us to make the assumption that if any instruction from the basic block executes, the rest of them will as well. Given concerns about code

obfuscation which may render this assumption to be false, this optimization may be easily rolled back to observing the executed code at the instruction level. In addition, we are able to observe all aspects of any detected direct branch. Statically, within instrumentation routines, direct branches can be identified along with their fallthrough and target addresses, the latter of which must be specified directly. The remainder of target addresses are only known dynamically, along with whether or not the instruction actually executed or not, which requires the use of analysis routines.

Beyond monitoring the execution of a binary, Pin allows for the modification of the program's context, which includes the register values currently stored in the processor. This ability provides the opportunity to change the `eip` register, and in turn the direction at which the code is executed. Given the task of forcibly changing a program's execution to explore an EEP, the virtual machine component uses Pin to first detect the presence of the branch that corresponds to the given EEP. At this time, the program's context may be modified to force either the target or fallthrough address to be the next instruction evaluated, which effectively explores the code at the EEP.

4.1.2 Dynamic Taint

Prior work [30] has considered the phenomenon known as *path explosion*, where there might be too many code entry points to practically evaluate within a multi-execution approach. Similar to this work, we utilize dynamic taint tracking as a method to reduce the number of entry points considered when their total count is too

high. Through taint analysis, if a data flow connection exists between a *taint source* and a branch, there is a high probability that this branch is correlated to the source. In this case, taint sources are Windows API functions that can allow evasive malware to hide their behavior; example functions include `GetUserName`, `GetCursorPos`, and `PathFileExists`. A connection between the use of such functions and the program branches can be established through first identifying the instructions that set the flags that are used when a branch's condition is evaluated; such `x86` instructions are commonly `test` or `cmp`. Given these instructions, their operands may be checked to see if they correspond to a tainted value. If this were to be the case, any use of the flags set by the instruction by a branch will cause it to be tainted, and hence a potential candidate for evaluating at a higher priority compared to other branches. With regards to DVasion's implementation, a simple heuristic is used given that branches that correspond to a `test` or `cmp` instruction are typically executed next, which allows for an easy association of these comparison instructions to branches.

To achieve this, we have utilized an instruction-level data tracking system built on top of Pin called `libdft` [37]. Built for Linux and `x86`, it utilizes the knowledge of its system calls and `x86` instructions to either propagate or create tainted data. The use of this system comes at a cost, as the average overhead is 3.65x; it is clear that this can be a powerful tool if used in a limited fashion given the overhead.

Modifying `libdft` to work on Windows requires a fundamental understanding on how the system works to tweak it in the right manner. This system essentially consists of two components: taint propagation through `x86` instructions, and taint creation through system calls. Given that DVasion runs on `x86`, only minor mod-

ifications were made to this component to allow for it to compile on Windows; furthermore, these edits have allowed for a more up-to-date version of Pin to be used with libdft, which matches the version used to create DVasion’s virtual machine component. The second component is where the majority of the modifications were made. Clearly, we cannot use the Linux system call table on Windows, and an equivalent table is not officially available through Microsoft. In place of this, we have instead written wrapper functions for 79 Windows API functions that could be used by evasive malware; these identified functions could be used for gathering information about the host computer or connected networks, checking for file or process existence, and checking the current time. Through these wrapper functions, we were able to successfully taint any relevant function arguments or returned value. To the best of our knowledge, we believe we have developed the first functional port of libdft from Linux to Windows.

4.2 Host Machine Component

The host machine component is responsible for orchestrating the execution of DVasion through performing a diverse range of different tasks that have been enumerated below. These requirements have directed us towards the use of an analysis system and a virtualized environment to observe new behavior.

1. **Behavior discovery.** The discovery of observable behavior that occurs during the execution of the binary; this will allow for new program behavior to be identified through the multi-execution analysis.

2. **Parallel, sandboxed evaluation.** The support of simultaneous exploration of a program’s EEPs and a protected environment that easily allows for this.
3. **Centralized control.** The control of the array of virtual machines utilized by DVasion and the maintenance of a global program state.

4.2.1 Cuckoo Sandbox

We chose to utilize Cuckoo Sandbox 1.2 [3], a malware analysis system, as it provides two significant contributions towards satisfying the enumerated goals for the host machine component. First, it allows for the dynamic detection of runtime behaviors through the use of signatures, written in the form of Python scripts. These signatures can detect a variety of behaviors, including simple checks for the establishment of a keylogger to the more complicated execution of a process that has had code injected into it. This detection is achieved through *cuckoomon.dll*, a DLL that is injected into a process that allows for the logging of its behaviors at runtime, which are then reported back to the main Cuckoo Sandbox process. Given any modifications to the control flow of a program through DBI, this may be used to detect any new behaviors found on this path, as they will differentiate based on what was seen previously in an unmodified execution.

Second, Cuckoo Sandbox natively supports the use of multiple virtual machines to evaluate programs at the same time. While one may consider this as an efficient way to process several different binaries at once in a traditional instance of Cuckoo Sandbox, this framework is essential to DVasion as it can allow multi-

ple paths to be explored at the same time for the same binary. Given the amount of paths we would like to explore in an allotted time, we have found this parallel analysis to be critical in reducing the overall runtime of DVasion.

It is important to note that Cuckoo Sandbox 1.2 has recently become out of date in favor of version 2.0. To use version 1.2 to the best of our ability, we integrated several new features, including a variety of new signatures from a fork of Cuckoo Sandbox 1.2 [38]. Chapter 6 reflects on this design decision and the future intentions to completely migrate from version 1.2 to 2.0.

4.2.2 KVM

We selected KVM to provide protected, virtualized sandbox environments to test the suspected malware samples. We found that KVM naturally supports fast snapshot restores and running multiple instances on the same machine, simultaneously. Conveniently, Cuckoo Sandbox natively supports KVM as an environment to evaluate programs in, so no modifications in that regard were required.

Despite containing malware within this protected environment to avoid any modifications made to the host computer, it would be ideal to allow the program the ability to connect to the Internet, as much of a program's behavior may require a valid, online connection to be present. As a compromise, we identified INetSim [39], a software suite that simulates internet services that can be used to trick the malware into believing that it is free to utilize the computer's internet connectivity. We found the virtual machines provided by KVM could easily support this extra consideration.

Through preliminary testing on a small set of 114 malware samples, we found that over 30% of them explored more code when INetSim was utilized through observing the execution’s code coverage; this confirmed the ability of an internet simulator to further a malware’s execution, and was used during the experimental testing.

Given the use of emulated environments to test malware in prior work, we feel that is important to reflect on how DVasion may perform if integrated with a system that uses emulated sandboxes, rather than virtualized ones. Recent anti-emulation strategies have proven effective in allowing evasive malware to hide certain portions of its code, including the use of leftover values in registers after a Windows API call [40]. In this case, the malware utilizes direct branches when comparing register values, whose outcome may be easily modified through DBI. Through examples like this, we are confident that the use of emulated environments can allow DVasion to detect evasive behavior in malware.

4.2.3 Driver

While Cuckoo Sandbox operates well as a stand-alone malware analysis system that may process a multitude of binaries through a traditional single-execution approach, it is clear that a main process, further referred to as the *driver*, is required to support the multi-execution method utilized by DVasion. This process, written in C++, interfaces with Cuckoo Sandbox through its available API, and is aware of any virtual machines that are available for use. Furthermore, this process will be responsible for maintaining the global state of DVasion as the algorithm progresses.

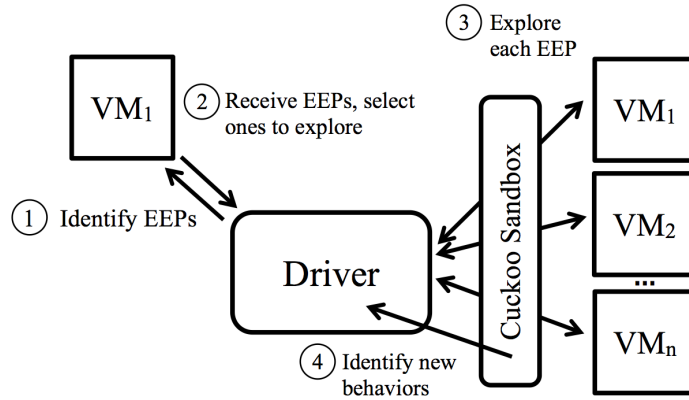


Figure 4.3: DVasion’s algorithm, including interaction with the virtual machines and Cuckoo Sandbox.

The driver essentially serves as the lead throughout DVasion’s execution; the virtual machine component acts as the leader’s helper as it processes a certain task within the virtual machine. To help capture how this process ties together the fundamentals previously discussed, it’s algorithm steps have been enumerated below and may be visualized in Figure 4.3.

1. Give two available virtual machines the task to strictly run the binary under test and identify any EEPs found; there will be no modifications in these runs, and one of the virtual machines will also perform taint analysis to identify tainted branches.
2. Receive the EEPs identified in the non-taint task previously issued. The driver will observe, through what code has been executed, the EEPs that are worth exploring; static and dynamic analysis schemes can be used to reduce this effectively, including prioritizing tainted EEPs.

3. Each EEP that has not been visited is evaluated as a separate task within an available virtual machine.
4. Evaluate the report generated by Cuckoo Sandbox for each task that involved binary rewriting to visit an EEP. Accumulate new signatures and print them at the end along with the corresponding EEP information.

Step 2’s static and dynamic analysis schemes provide a vital way to reduce DVasion’s overhead in an intelligent manner. Statically, we are able to inspect the code at the basic block which corresponds to the address of an EEP when it is first discovered at runtime. Through this, we are able to identify EEPs that will lead to a “safe” code segment, one that does not execute a malicious action or cause the program’s execution to go to additional unvisited code. If this is the case, we conclude that it is not worth investigating this EEP further and discard it. Following our assumption that maintaining a malware’s program state is not important when seeking evasive functionality, this strategy is safe.

Despite this static reduction in the EEP count, it may be the case that there are still too many qualified EEPs to explore; here, we use taint-analysis to only consider EEPs whose associated branch is based off a tainted value, as described in Section 4.1.2. While this reduces our ability to detect new behaviors that utilize Windows API functions in a way that we did not anticipate, this is an effective strategy to narrow down the EEPs to candidates that are likely to be hiding evasive code based on their taint relationship.

Through this mutli-step process, DVasion is able to successfully find new be-

havior that may be lurking at an EEP.

Chapter 5: Experimental Results

In this section, we discuss the results of applying DVasion on both specific malware families and a large, diverse set of malware. The application of DVasion to specific case studies confirms that it observes the evasive behavior that has been documented in prior work, whereas the large dataset analysis shows the extent at which DVasion can derive new behavior from an array of binaries.

The following tests were run on a machine with two AMD Opteron Processor 6212 (4 cores per socket, 2 threads per core), with 128 GB of RAM. 32 KVM instances, with Windows 7 installed, were created and used for parallel EEP evaluation, where each machine was configured to run on a single core with 1 GB of RAM. To reduce the overhead of restoring from a snapshot, along with the general use of multiple virtual machines at the same time, these machines were placed in RAM. All tasks were limited to 45 and 35 seconds for the first and additional executions, respectively, with the exception of the taint-analysis pass that used 90 seconds to account for its higher overhead imposed on the binary it was instrumenting. *Severity* values accompany each Cuckoo Sandbox signature to indicate, on scale from 1 to 3, how malicious that behavior is. Only a couple of severity values used were decremented by 1, as we felt it was an appropriate response to how common they

were found among all programs. Lastly, our current experiments bypass a single level of evasion within the specific binary that is currently being evaluated; relaxing this constraint is explored in Section 5.1.1.

Cuckoo Sandbox will provide the mechanism to detect any malicious behaviors found when a branch's alternate path is explored; to ensure that we report on signatures that were derived solely because of this additional execution, we have independently run all evaluated binaries through a standard instance of Cuckoo Sandbox 1.2 that is only augmented through additional signatures used with DVasion. Here, these binaries were given 3 minutes to execute, with at most 8 of the KVM instances running at the same time to ensure the programs were able to progress and show any behavior that would have been observed through this traditional, single-execution approach. This also addresses any concern that the evasive behavior observed through DVasion is in fact caused by the detected presence of Pin, and that this behavior would have executed if Pin was not used.

5.1 Specific Malware Case Studies

5.1.1 Dyre

The Dyre Trojan, first spotted in June 2014, has performed a multitude of attacks across large corporations including Citigroup, JPMorgan Chase, and Bank of America [41]. This malware exhibits evasive behavior through checking the number of cores present in a system, as discussed in Section 3.1.

Individually, the virtual machines used for DVasion will fail to detect the

additional, evasive behavior, given that each KVM instance has been allocated a single core for testing. However, through the multi-execution analysis performed by DVasion, we were able to discover this behavior along with what it was hiding: fingerprinting of the host environment and the creation of an executable. Though DVasion targets the exploration of a specific binary through its EEPs, we expanded its jurisdiction in this case study to include the spawned executable. Interesting enough, this binary too contained evasive behavior, specifically additional fingerprinting of the host environment and the creation of a service called “Google Update Service”. A summary of these discoveries may be found in Table 5.1, and strongly parallel the alert issued by US-CERT regarding Dyre’s analyzed behavior [42].

Signature	Severity	Original	DVasion
Installs itself for autorun at Windows startup	3	X	X
Collects information to fingerprint the system	3		X
Creates a Windows executable on the filesystem	2		X
Process creates service object(s) (googleupdate)	3		X

Table 5.1: Behavior detected in the original and DVasion analyses of the Dyre malware. An ‘X’ indicates that this signature was found in either the original or DVasion report.

5.1.2 Carbanak

Multiple studies have been performed on the Carbanak APT, which has created havoc for banking and financial institutions worldwide, with estimated losses totaling upwards of 1 billion USD [43]. The Carbanak malware targets financial accounts and ATMs, obtaining access through spear phishing emails that led to the installation of a remote backdoor that allowed for access to infected machines.

As indicated in a Lastline [44] study of the 74 Carbanak samples provided in the Kaspersky report [43], they found 17% of binaries were found to demonstrate evasive behavior including the detection of a virtual sandbox or going to sleep. These results provide a baseline target that DVasion should be able to match, which can prove the developed system's effectiveness.

Through the file hashes provided at the end of the Kaspersky report, we were able to obtain all 74 binaries and applied DVasion to each. The results strongly parallel the Lastline findings with signatures that encompass the evasive behavior they cited at a higher detection rate: we found this behavior to exist in over half of the programs. These results have been summarized Table 5.2 and attest to DVasion's ability to discover accurate, evasive behavior.

5.2 Large Malware Dataset Study

Having successfully matched prior analyses performed on evasive malware with the results obtained through DVasion, we have found confidence in our system's ability to identify evasive behavior. Seeking to apply this to a general set of malware,

Signature	Severity	Occurrences
Installs itself for autorun at Windows startup	3	35
Creates a Windows executable on the filesystem	2	32
Collects information to fingerprint the system	3	32
A process attempted to delay the analysis task	2	31
Accesses web history	2	29
Likely virus infection of existing system binary	3	25
A process created a hidden window	3	1

Table 5.2: New behavior from the Carbanak dataset.

we obtained over 32,000 samples from VirusTotal [45], randomly selected 1,000 of them, and applied DVasion to each. The VirusTotal API allowed us to specify binaries from 2014 and 2015, as well as those that were a 32-bit executable; these qualities were verified independently by us.

Through this analysis, we found that 167 of the 1,000 binaries (16.7%) had new behavior discovered through the multi-execution approach utilized by DVasion; of these programs with new behavior, 72 (43.1%) of them did not have any signatures detected before. This new behavior consists of 252 instances of new severity-3 signatures detected, and 157 instances of new severity-2 signatures; the specifics of these new behaviors may be found in Table 5.3. On average, DVasion took 234 seconds to perform its analysis and generate a report, which is a major success given that such analyses may take on the scale of hours by a human analyst.

We have found two metrics to be especially helpful in collectively understanding the new signatures found. First, one may consider the severity values that correspond to the signatures in Cuckoo Sandbox. Given the detection of a signature with a severity value higher than what has previously been seen can be helpful in understanding the program's malicious capabilities better. Table 5.4 captures this metric as it covers all 6 possible largest severity transitions possible, including not detecting any signatures (highest severity value would be zero) and finding a signature that either has a severity value of 1, 2 or 3.

Second, the additional behavior found may be captured by the sum of unique signature severity values; this involves considering two sets: the signatures found without DVasion, and the signatures found with DVasion. If you were to sum up

the corresponding severity values of the signatures found in each set and compare the two, a significant increase could indicate the presence of extensive new behavior. Of the 167 binaries that had new behavior discovered, their original sum of severity value was a 4; through DVasion, this value was improved to 10. This indicates that when DVasion is able to discover new behavior, it typically discovers multiple number signatures given that signatures usually have a severity value of 2 or 3.

Signature	Severity	Occurrences
Installs itself for autorun at Windows startup	3	96
Collects information to fingerprint the system	3	63
Accesses certain keys, possibly to modify browser security settings	2	51
Steals private information from local Internet browsers	3	44
Accesses web history	2	44
Opens keys used to modify proxy settings	2	19
Creates a Windows executable on the filesystem	2	17
Accesses keys, possibly to collect information about installed applications	2	15
Checks the version of Bios, possibly for anti-virtualization	3	12
Detects the presence of Wine emulator via function name	3	11
Detects Sandboxie through the presence of a library	3	11
A process created a hidden window	2	8
Likely virus infection of existing system binary	3	7
Executed a process and injected code into it, probably while unpacking	3	2
Starts servers listening on a specific port	2	2
Installs an hook procedure to monitor for mouse events	3	1
Detects virtualization software with SCSI Disk Identifier trick	3	1
A process attempted to delay the analysis task	2	1
Harvests information related to installed mail clients	3	1
Queries information on disks, possibly for anti-virtualization	3	1
Attempts to remove evidence of file being downloaded from the Internet	3	1
Checks for the presence of known devices from debuggers and forensic tools	3	1
Network anomalies occurred during the analysis	2	1

Table 5.3: New behavior from the large dataset.

Before	After	Occurrences
0	1	0
0	2	2
0	3	70
1	2	0
1	3	0
2	3	26

Table 5.4: Malware that showed an increase in its highest signature severity value.

Chapter 6: Future Work

Our system confirms that program state does not need to be maintained to expose additional, new behavior through a multiple-execution analysis of program binaries. An unintended consequence of such a decision includes the vulnerability to unintentional (e.g. due to inconsistent program state) or intentional (e.g. divide by zero after an anticipated rewrite) program crashes that may prevent the analysis system from observing additional behavior in a program. Initially, one may consider the incorporation of methodologies that maintain program state, but such systems are imperfect for a variety of reasons, as cited in the related work within [Section 2.4.2](#). Future work can extend our policy of not maintaining state, and simply ignore program crashes.

Our system utilizes Cuckoo Sandbox 1.2, which was the current version during DVasion’s development. Since then, this has been rendered out-of-date with version 2.0. While additional features were integrated to our current setup of Cuckoo Sandbox, not all new features could be easily ported back to 1.2. We anticipate on updating the Cuckoo Sandbox instance used with DVasion to 2.0 soon, which should usher in a variety of new features and previously incompatible signatures to detect additional evasive behaviors within the binaries.

While the VirusTotal dataset utilized for our experimental results has proven to yield a diverse range in malware samples from different families, these samples lack the ability to expose DVasion to real, zero-day attacks. Even though such samples are comparatively difficult to obtain, we plan on obtaining such program binaries through the integration with a honeypot. Furthermore, this test will confirm DVasion’s ability to handle the throughput of testing all binaries that are captured from this real system.

Similar to previous multiple-execution analysis systems, we share the limitation that indirect branches are not considered as a source of evasive entry points, which may potentially restrict our ability to view a more complete picture of the binary. Unfortunately, this class of branches can be used evasively, for example, through the use of a system’s clipboard count obtained by the the Windows API `CountClipboardFormats` to create a multitude of locations the control flow of a program may follow [46]; this evasion exploits the intuition that a real system will likely have contents copied to the system’s clipboard, whereas a sandbox is likely absent of such evidence of human interaction. Indirect branches are notably more complex to analyze the direct branches given that they may cause a program’s control flow to travel to more than two possible locations; for direct branches, we only had to consider the fallthrough and target addresses as potentially EEPs. Our future work envisions the incorporation of features like jump tables, when present, which may allow for all possible outcomes from an indirect branch to be discovered and considered as EEPs.

Most recently, we have investigated ways to lower DVasion’s overhead through

using a lighter test environment known as a container. DVasion uses a dedicated virtual machine instance for each EEP that is explored in its multiple execution analysis, which essentially translates to multiple Windows 7 instances running simultaneously. Containers seek to isolate processes in user-space, which we believe could be a way to allow multiple EEPs to be visited in parallel within the same virtual machine. We hope that the integration of such a testing environment can allow for more EEPs to be evaluated at the same time.

Chapter 7: Conclusion

In this paper, we have described DVasion, an effective tool in uncovering hidden behavior within evasive malware. Through a multi-execution approach, our solution is able to explore additional code segments that previously would not have been executed in a traditional, single-execution system. This expansion in code execution directly translates to the possibility of uncovering previously unseen behavior, which is detected through the behavioral monitoring component of Cuckoo Sandbox. In order to modify the program at run-time to reach these additional branch outcomes, dynamic binary instrumentation provided through Intel Pin proved to be essential. The overhead of such a system was heavily considered throughout its development; helpful static and dynamic methods, including dynamic taint tracking, proved to be an effective manner in reducing the additional code that needed to be explored.

The results achieved through DVasion strongly indicate that it is successful in discovering hidden, malicious behavior in malware. Through strong parallels in DVasion's results with existing work on evasive malware, we are able to gain a significant amount of confidence in the accuracy and potential such a system has dealing with this variant of malicious binaries. Through an application of our system

to a dataset of 1,000 malware instances, we found 167 contained hidden behavior. This success validates our design decision to not maintain program state, given that if this were not the case, our results would be absent of such a frequent, diverse set of new behaviors.

Bibliography

- [1] Nsa chief: Cybercrime constitutes the “greatest transfer of wealth in history”. "<http://foreignpolicy.com/2012/07/09/nsa-chief-cybercrime-constitutes-the-greatest-transfer-of-wealth-in-history/>". Accessed: 2015-11-18.
- [2] Taking the cyberattack threat seriously. "<http://online.wsj.com/article/SB10000872396390444330904577535492693044650.html>". Accessed: 2015-11-18.
- [3] Claudio Guarnieri, A Tanasi, J Bremer, and M Schloesser. The cuckoo sandbox, 2012.
- [4] Anubis: Analyzing unknown binaries. <http://anubis.iseclab.org/>.
- [5] Joe sandbox. <https://www.joesecurity.org>.
- [6] Engin Kirda. Uncloaking the dark arts of evasive malware. "<https://securityintelligence.com/uncloaking-the-dark-arts-of-evasive-malware/>", November 3 2014. Accessed: 2016-04-06.
- [7] The threat of evasive malware. "<http://www.lastline.com/papers/evasivethreats.pdf>". Accessed: 2015-11-18.
- [8] Thomas Fox-Brewster. Netflix is dumping anti-virus, presages death of an industry. "<http://www.forbes.com/sites/thomasbrewster/2015/08/26/netflix-and-death-of-anti-virus/#5fae7bbc3256>", August 26 2015. Accessed: 2016-04-06.
- [9] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.
- [10] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

- [11] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 91–100. IEEE, 2004.
- [12] Felix Freiling. Toward automated dynamic malware analysis using cwsandbox.
- [13] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. *TTAnalyze: A tool for analyzing malware*. na, 2006.
- [14] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.
- [15] Kevin Borders, Xin Zhao, and Atul Prakash. Siren: Catching evasive malware. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.
- [16] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.
- [17] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.
- [18] Christopher Kruegel. Full system emulation: Achieving successful automated dynamic analysis of evasive malware. In *Proc. BlackHat USA Security Conference*, 2014.
- [19] Nathanael R Paul. *Disk-level behavioral malware detection*. PhD thesis, University of Virginia, 2008.
- [20] Madhusudhanan Chandrasekaran, S Vidyaraman, and S Upadhyaya. Spycon: Emulating user activities to detect evasive spyware. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE International*, pages 502–509. IEEE, 2007.
- [21] Jedidiah R Crandall, Gary Wassermann, Daniela AS de Oliveira, Zhendong Su, S Felix Wu, and Frederic T Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *ACM Sigplan Notices*, volume 41, pages 25–36. ACM, 2006.
- [22] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296. ACM, 2011.

- [23] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS*, 2010.
- [24] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.
- [25] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 403–412. ACM, 2011.
- [26] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 71–80. ACM, 2015.
- [27] Dhilung Kirat and Giovanni Vigna. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 769–780. ACM, 2015.
- [28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [29] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.
- [30] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.
- [31] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.
- [32] Jeffrey Wilhelm and Tzi-cker Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection*, pages 219–235. Springer, 2007.
- [33] B. Livshits, B.G. Zorn, C. Seifert, and C. Kolbitsch. Execution of multiple execution paths, July 4 2013. US Patent App. 13/339,322.
- [34] C. Kolbitsch, P.M. Comparetti, and L. Cavedon. Methods and systems for malware detection based on environment-dependent behavior, October 23 2014. US Patent App. 13/866,980.

- [35] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.
- [36] Derek Bruening. Qz: Dynamorio: Dynamic instrumentation tool platform.
- [37] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, volume 47, pages 121–132. ACM, 2012.
- [38] Brad Spengler. cuckoo-modified. "<https://github.com/spender-sandbox/cuckoo-modified>", 2015.
- [39] Thomas Hungenberg and Matthias Eckert. Inetsim: Internet services simulation suite, 2013.
- [40] Second Part To Hell. Dynamic anti-emulation using blackbox analysis., October 2 2011.
- [41] Protecting against the dyre trojan: Don't bring a knife to a gunfight. "<https://securityintelligence.com/protecting-against-the-dyre-trojan-dont-bring-a-knife-to-a-gunfight/>". Accessed: 2016-03-09.
- [42] Alert (ta14-300a): Phishing campaign linked with “dyre” banking malware. "<https://www.us-cert.gov/ncas/alerts/TA14-300A>". Accessed: 2016-03-21.
- [43] Carbanak apt: The great bank robbery. "https://securelist.com/files/2015/02/Carbanak_APT_eng.pdf". Accessed: 2016-03-09.
- [44] Carbanak malware - ninety five percent exhibits stealth or evasive behaviors. "<http://labs.lastline.com/ninety-five-percent-of-carbanak-malware-exhibits-stealthy-or-evasive-behaviors>". Accessed: 2016-03-09.
- [45] Virus Total. Virustotal-free online virus, malware and url scanner, 2012.
- [46] Does dyre malware play nice in your sandbox? "<http://labs.lastline.com/dyre-malware-does-it-play-nice-in-your-sandbox>". Accessed: 2016-03-09.