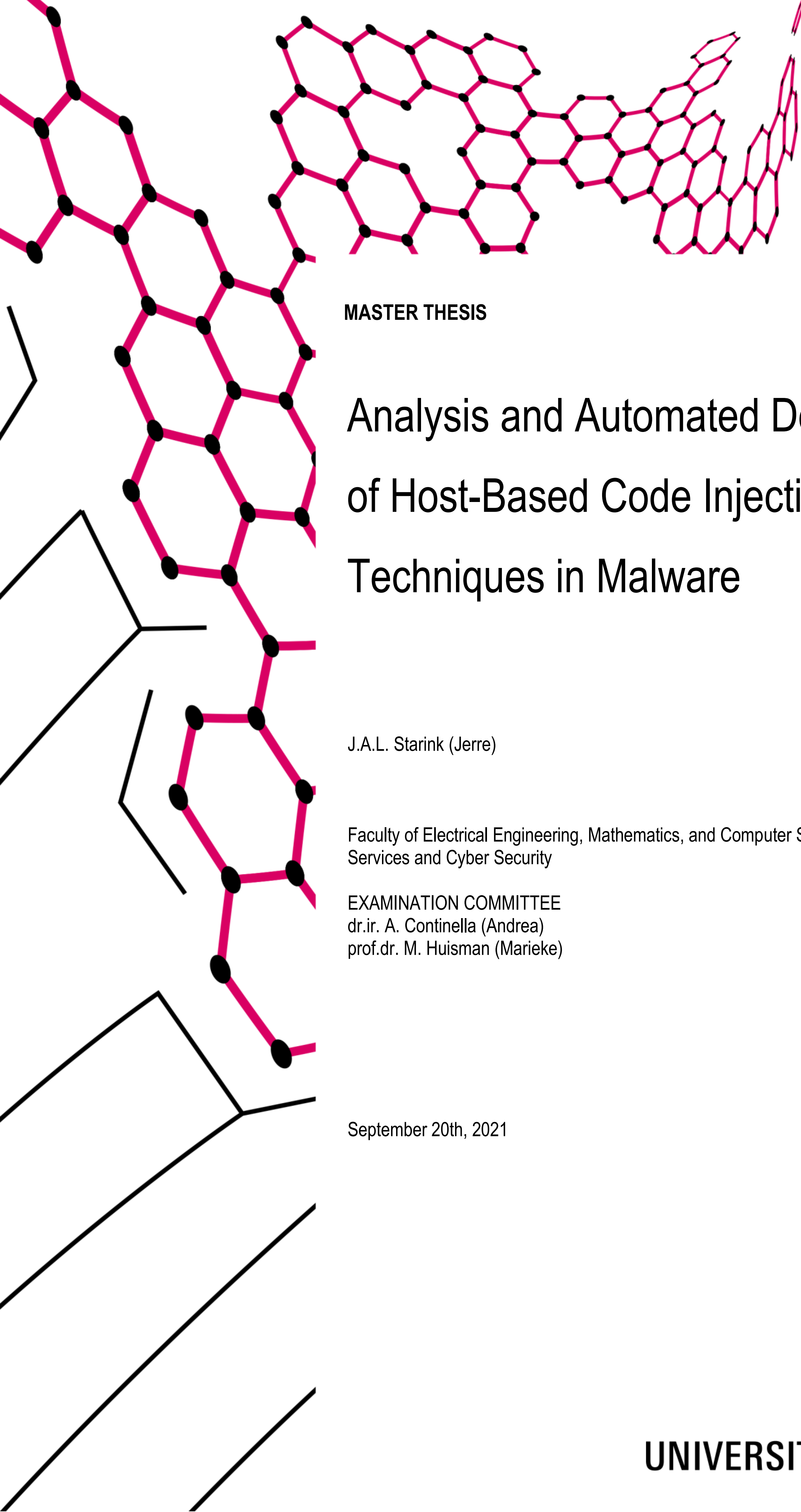

MASTER THESIS

Analysis and Automated Detection of Host-Based Code Injection Techniques in Malware

J.A.L. Starink (Jerre)

Faculty of Electrical Engineering, Mathematics, and Computer Science
Services and Cyber Security

EXAMINATION COMMITTEE
dr.ir. A. Continella (Andrea)
prof.dr. M. Huisman (Marieke)

September 20th, 2021

UNIVERSITY OF TWENTE.

Abstract

For malware to be successful, it should stay undetected by anti-virus software for as long as possible. One method for avoiding detection is the use of code injection, which is the process of injecting code into another running application. Despite code injection becoming one of the main features of today's malware, there has been a general lack of a systematic approach in analyzing and detecting the use of it. In this research, we conduct a study on well-known methods for performing code injection, and propose a taxonomy that groups these methods into classes based on common characteristics. We then introduce *Behavior Nets*, our novel modelling language that we use to express these methods in terms of observable events. We continue by implementing a system that uses these models to collect empirical evidence for the prevalence of code injection in the malware scene. Our experiments suggest that at least 11.15% of malware between 2017 and 2020 performs some type of injection. They also show that Process Hollowing is the most commonly used technique, but that this trend is slowly shifting towards other, less traditional methods.

Keywords: Malware, detection avoidance, code injection, software reverse engineering, dynamic analysis, modelling language, black box testing.

1 Introduction

In the world of cyberspace, one of the main driving forces that make cyber security incidents a reality, is the use of malware. The term *malware* is a conjuction of the words *malicious* and *software*, and is an umbrella term for software that is intentionally designed to cause harm. There are many types of malware, and each type has a different profile of behaviors that they may exhibit. For example, some malware samples might steal or destroy important files stored on the disk, while others will steal important information such as login credentials instead. Typically, the ultimate end goal of malware developers is to profit financially [9, 21, 38, 65].

Malware has existed for a long time, and has become infamous in today's society. One of the first instances of malware that gained significant media recognition was the Morris Worm, created by Robert Morris in 1988 [55]. Since then, many other malicious programs were developed, and the number of malware samples is growing steadily [6].

To fight against the malware epidemic, several parties have started developing software that is specifically designed to detect malware stored or running on the protected machine. These anti-malware solutions have gained a lot of popularity over the past years, and are nowadays installed by default on virtually every general purpose computer.

For malware to be successful, it is therefore in the creator's best interest to make sure that it stays under the radar of these anti-malware solutions for as long as possible. One of the techniques that can be used to avoid detection is known as *code injection*. Code injection can be defined as the process in which an application injects pieces of its own code into another running program. This running program is then tricked into executing the injected code, making it do something it was not originally intended to do [12, 13]. By extension, if a malicious program copies its malicious code into a legitimate application, it is not the original malware itself that exhibits the malicious behaviour, but rather the application that was previously considered to be benign. As a consequence, scanning an executable file existing on the disk for suspicious code might not be sufficient, making the task for automating threat detection systems significantly more involved.

Currently, detection of the presence of code injection is either done by manually reverse engineering a sample and looking for code constructs that would indicate this behavior, or with the help of heuristics such as testing for known byte patterns or used system calls. However, there are various ways of performing code injection, and it is expected that new, more sophisticated methods will be discovered and implemented in the future. Furthermore, the rise of the amount of computers that people own, combined with the increase in malware prevalence, render both manual analysis and the use of these relatively primitive heuristics as insufficient for reliable detection of code injection. There is a need for a better, more

fundamental understanding of what a code injection entails, as well as a more systematic and more scalable method for detecting this type of behavior.

1.1 Contributions

In this research, we conduct a systematic study on the most well-known methods that can be used to achieve code injection. We do this by collecting implementations for every technique, and test them to see if they are still working on software and hardware that is commonly used at the time of writing this paper. We then continue by comparing every technique to each other, and identify reoccurring features and characteristics. From this, we derive a more fundamental understanding of code injection, and propose a categorization of all the studied techniques based on these common characteristics.

After building up this classification, we move on by developing a modelling language that allows us to build up formal representations for every technique. We call these models *Behavior Nets*, and they express the techniques in terms of observable events and the dependency relations between them. We then implement an automated system that uses these behavior nets to determine whether an arbitrary sample uses one of the fingerprinted code injection techniques. Finally, we evaluate our system by running it through a data set of 3075 real world malware samples, and show that not only that our system works, but also how prevalent the use of code injection is in the malware scene as of the time writing this paper.

In short, the main contributions of this paper can be summarized in the following:

- **A Taxonomy of Code Injection:** We conducted a survey on 17 different code injection techniques, and propose a taxonomy which classifies the different techniques based on a set of identified common traits.
- **The concept of Behavior Nets:** A modelling language that can be used to detect certain types of behavior exhibited by a sample in a black-box manner.
- **A Code Injection Detection System:** An implementation of a system that detects the presence of code injection in a malware sample.
- **An Assessment on the Prevalence of Code Injection:** We have examined a set of 3075 malware samples, and determined the prevalence and distribution of different code injection techniques in the wild.

We have made our implementations, as well as our test files for the studied code injection techniques, open source[1] for the sake of open science.

1.2 Paper Structure

The remainder of this paper is organized as follows. We start off by introducing the topic of code injection in more detail, and cover certain concepts in the area of reverse engineering in Section 2. We then continue with a survey on state-of-the-art code injection, and provide a classification of the different existing techniques based on common characteristics in Section 3. In Section 4, we move on to describing the process on how we detect these types of behaviors in a given sample. We continue by outlining the architecture of our test environment that implements this type of detection system in Section 5, and present our findings in Section 6. We discuss our results in Section 7 and relate them to previously conducted research in Section 9. Finally, we conclude by summarizing what was done in our research in Section 10.

2 Background

Since the focus on this paper lies in studying and detecting the presence of code injection techniques within samples of malware, it is important to understand the fundamentals of some of the concepts in this field. In this section, we will introduce the notion of what a code injection entails, and explore how it can be used legitimately as well as maliciously. Furthermore, since one contribution of this paper is an automated system for detecting these types of behaviors, we will also go over the fundamental concepts in the world of program analysis, and what kinds of

[1] `https://github.com/jstarink/code-injection`

strategies can be employed to infer certain types of behavior in an application.

2.1 Code Injection Techniques

As briefly stated in the introduction, a code injection can be defined as the act of injecting code into another running process. The basic steps usually involve finding a *victim* process, selecting some existing executable memory or dynamically allocating some new memory in this process, copying over the new code into this memory, and then making sure the victim process executes it. The goal of code injection is usually to ensure that the injected code is executed in the context of the victim process, making the victim process do something it was not originally designed to do.

2.1.1 Legitimate Use-Cases

One of the main reasons someone might want to inject code into another process is for debugging purposes. A *debugger* allows a developer to step through the compiled code of their own software, and observe the state changes that their program goes through by inspecting the program's internal memory. Many debuggers rely on placing *software breakpoints* into the target application. Software breakpoints are small temporary changes in the code that signal an interrupt. This effectively pauses the execution of a program, leaving the developer with time verify whether the program is doing as was expected. Examples of software breakpoint implementations are the `int3` instruction on the Intel x86 platform [36, p. 457] and the `bkpt` instruction on ARM [1].

Another legitimate use-case for code injection techniques is to increase software compatibility with the help of *shims*. As time progresses, the operating systems that people run on their machines evolve. Changes in the operating system's code might range from small bug fixes to complete API redesigns. Software that relies on old legacy designs might therefore not be compatible with newer versions of the operating system. An API might simply not exist any more, or may exhibit different behavior after the version update. A shim infrastructure allows for redirecting API calls to *shim code* on a per-process level. By doing this, the shim can masquerade as the old API, and make up for the changes that were introduced in the version update, by calling the new or appropriate APIs instead. Examples of shim infrastructure implementations are the Microsoft Application Compatibility Toolkit (ACT) for Windows [45], and the `LD_PRELOAD` environment variable on various Linux distributions [5].

2.1.2 Malicious Use-Cases

As alluded before, injecting code into another running process is a very effective way to hide the true behavior of an executable file. For this reason, code injection has been prevalent in many different malware families, each using their own variant of performing the injection of their malicious code in another running process. Since the malicious code is not executed by the malware anymore, the original sample might seem benign at first glance, and therefore bypass all kinds of detection mechanisms implemented by anti-virus software. This way, malware can easily stay undetected for long periods of time.

One famous example is the Stuxnet worm, which was first seen in 2009. Stuxnet used a technique called DLL injection, where the target process is tricked into loading a custom (malicious) dynamically loaded library. By spawning a new thread in the victim process (e.g. using the `CreateRemoteThread` function) with carefully chosen starting parameters, it is possible to let the process call the `LoadLibrary` function with the path to the malicious DLL with very few changes in the original memory of the process [25]. Using this technique, Stuxnet was able to infect approximately 100,000 machines by September 2010 [34].

Another example is the ZeroAccess botnet, which was discovered around 2011. By abusing certain features of the Asynchronous Procedure Call (APC) queue of running threads, ZeroAccess successfully injected and ran code in the context of `explorer.exe` and `svchost.exe`, two known core processes of the Windows operating system. It was estimated that the botnet was installed around 9 million times in 2012 [65].

2.2 Malware Analysis and Reverse Engineering

Since malware is a special form of software, examining malware samples is a special case of software analysis. The challenge here is that malware is often shipped as a compiled binary, and does not include source code that we can look into easily. This means that our options for inferring something about the behavior of such a sample are somewhat limited. In fact, if we want to have any success in recognizing any type of nefarious behavior, we are forced to apply some form of Software Reverse Engineering. Software Reverse Engineering (SRE) is the process of analyzing a software system, with the goal to recover (parts of) the original design or implementation [19]. Typically, SRE is used to recover lost source code of an application that has been in development for a long period of time. However, it has been used by many security experts to analyze and neutralize many types of malware as well.

In the following, we will go over the basic concepts of the two main paradigms in software analysis, called *static* and *dynamic analysis.* For both paradigms, we will put them in the context of SRE, and list certain advantages and challenges when applying them to malware analysis.

2.2.1 Static Analysis

Static analysis is a form of program analysis that stems from the fundamental principle that computers are deterministic machines. Given the same input state and set of instructions to execute, a program or algorithm always produces the same result, regardless of the number of repetitions. Therefore, if a program were to exhibit a certain behavior at run time, it must mean that this behavior is somehow encoded in its instructions. Let us define static analysis as the following:

Definition 1 ***Static analysis*** *is any form of program analysis that makes an assessment on the program's behavior solely based on the code of the input program, without actually running the program itself [17, 31].*

Static analysis often relies on analyzing the original source code of the program. As mentioned before, usually in the context of malware analysis, only compiled binaries are available and source code is not included. However, we can often still make use of this methodology if we perform some additional steps. For example, by disassembling the input file, it is possible to split up the binary code into *basic blocks*, and reconstruct a *control flow graph* that encodes all possible paths that the program might take. Let us introduce these two concepts more formally:

Definition 2 *A* ***basic block (BB)*** *in a program is a sequence of instructions that only has incoming branches at the entry, and only has outgoing branches at the exit of the block [22, p. 231].*

Definition 3 *A* ***control flow graph (CFG)*** *of a program P is a directed graph $G = (V, E)$ such that every $v \in V$ represents one basic block in P, and for the basic blocks $s, t \in V$ there exists an edge $(s, t) \in E$ if and only if s can transfer control to t [22, p. 231].*

An example CFG can be found in Figure 1. In this CFG, the basic blocks contain disassembled x86 code of an if-statement. Depending on the value of the `eax` register, the program either jumps to `block2` and call the function `foo`, or fall through into `block1` and call `bar` instead. However, no matter which path is taken, the program will always end up in block `block3` that invokes the function `baz`, and continue execution from there on.

From these CFGs, higher abstractions can be derived, such as a call graph (which encodes the relationship between different functions), and sometimes even source code that is semantically equivalent to the original [17, 18]. Once these types of models are reconstructed, the same techniques used in traditional static analysis can be performed to infer certain properties on a program's behavior.

Advantages The main advantages of static analysis in the context of malware examination is evidently that by definition it does not require the malware to be executed. This ensures that the environment of the researcher does not get contaminated with infections while performing the analysis.

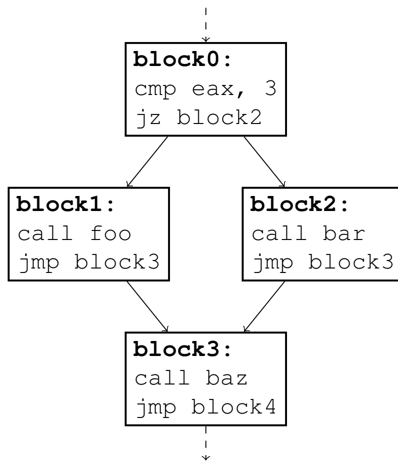

Figure 1: An example subgraph of a CFG implementing an if statement.

Furthermore, since programs can be modelled using control flow graphs, formal proofs can be derived from the structures within the graph, as all code execution paths can be considered. Often, these kinds of problems can also be rewritten as code optimization problems, which have been widely used in the field of compiler theory [22, 30] and formal software verification [29]. Since there has been a lot of research put into these fields, static analysis benefits a lot from the advances that are made, and can therefore be a very powerful tool for malware analysis.

Challenges One of the main challenges that reverse engineers face while performing static analysis, is dealing with *code obfuscation*. The main goal of code obfuscation is to transform the original program into a new one that is semantically equivalent in execution, but very hard to understand for a human reverse engineer [11]. One reason for doing this is to protect the code from being stolen, or to prevent changes being made [57, 60]. Transformations that are often applied to the original program include but are not limited to; symbol renaming or removal, encryption of constants such as strings, control flow obfuscation, dead code insertion, or even transpiling the original code into a different language using a virtual machine [16, 28, 35, 40, 43, 57, 67]. Obfuscation is an effective way to increase the complexity of a program, and is therefore also proven to be successful way to combat the process of reverse engineering it. For this reason, malware developers also have been using it to hide their malicious code, and use it as a detection evasion technique [59, 60].

Next to obfuscation, programs can also be compressed or encrypted using what is known as a *packer*. In such a case, upon execution of the application, the program first reconstructs the original binary code from the compressed or encrypted data, and then jumps into this dynamically allocated code [42]. While one of the main goals for software packing is to simply reduce the size of the final binary, it can also be used as an anti reverse engineering technique [66, 64, 59]. As the original code is not put in a readable format any more, it renders standard methods for extracting basic models, such as a control flow graph, completely useless. For this reason, malware authors have used it to not only lower the size of their payload, but also to circumvent detection by anti-virus software. Packers that are specifically built for evading anti-virus detections are sometimes also referred to as *crypters* [10, 14].

2.2.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis works under the assumption that if a program is performing some kind of operation, its effects should be observable in the environment, regardless of how complicated the implementation is. The application is often treated as more of a black box, and the focus is put more on *what* the end result is, rather than on *how* exactly it achieves this result. Let us define dynamic analysis as follows:

Definition 4 ***Dynamic analysis*** *is any form of program analysis that makes an assessment on the program's behavior, by executing the program and directly observing how it affects the internal state of the program, or the environment it runs in [31].*

Side effects produced by a program can be observed in many different ways. For example, the analyst can get a rough overview of the program's behavior by

monitoring the calls it makes to system libraries or the kernel at run time. Another way is to look for changes in the computer itself, such as changes in the file system or registry. Other programs will interact with remote hosts over the internet, and will open network sockets and transmit large chunks of data through them. In the context of malware analysis, these kinds of events can be very important in determining what kind of damage it inflicts on the underlying system.

Advantages One of the main advantages of dynamic analysis, is that it can be very computationally cheap in comparison to static analysis. As alluded to in the previous section, a lot of the indicators do not require deep analysis of the code, as is the case with static analysis. Instead, most side effects can be directly observed from the environment, without even looking into the actual program itself. This bypasses a lot of the anti reverse engineering tricks, such as code obfuscation or packing, something that static analysis has trouble with.

Challenges Dynamic analysis does not come without challenges. One of the main limitations of dynamic analysis is that it is not guaranteed to explore the entire state space of a program. Rather, it heavily relies on single execution traces that a program produces every time it is ran. A program might exhibit different behavior the next time it is started, or only starts doing something after a certain criteria was met [20]. Furthermore, dynamic analysis often requires some form of preparation or instrumentation, which can introduce all kinds of technical problems which might affect the program's behavior [41]. An assessment on the behavior of a program that is fully based on dynamic analysis might therefore not be an accurate description of the actual behavior that a program would exhibit during normal execution.

In the context of malware analysis, these points are extra important. For example, some malware stays dormant for days before it starts exhibiting noticeable malicious behavior [38]. Dynamic analysis cannot run indefinitely, which raises the question; for how long should we run the program before we abort the analysis? Clearly, this is an undecidable problem: If dynamic analysis is set to stop execution after t seconds, there will always be a possibility for the existence of a sample that starts showing illicit behavior after $t + 1$ seconds.

Additionally, as an analyst it is important to remain completely unnoticed by the malware. There are many different approaches a program can take to detect that it is being observed by a reverse engineer. For example, the presence of a debugger program on the system can be verified in many different ways [63]. Furthermore, since it is in the analyst's best interest to not cause damage to their own machine, some form of sandboxing or virtualization is required as to not get exposed to any of the malicious behavior that the sample might exhibit. The problem with this is that existing technologies for hardware virtualization are not always accurate or necessarily built to be stealthy. A program could look for irregularities that instrumentation or a virtual machine might introduce as a result of ad-hoc code patches, or slow or incorrect emulation of hardware [41]. Once malware detects one of these artifacts, it can then decide to show "normal" harmless behaviour instead, such as exiting early or staying dormant. This might make the analyst believe the program is benign, whereas in reality it is not.

3 Systematic Study of Code Injection Techniques

Since we want to move towards a system that is able to detect the presence of code injection in an arbitrary sample, we require a more fundamental understanding of code injection itself. To get to this understanding, we conducted a survey on 17 most well-known state-of-the-art code injection techniques that are used in wild and are talked about a lot by people in the security community. The techniques where gathered by collecting various blog posts and technical reports that dissect malware samples in detail, and explain how these samples implement code injection. These reports were published by anti-virus companies, incident-response teams, as well as other people active in the security community.

For each technique, we either reimplemented it ourselves, or collected an existing open source implementation from code hosting websites such as GitHub. This way, we end up with a small set of samples that acts as a form of a *ground truth*, where each technique is represented by at least one sample for which we have the source code available.

We then continued by identifying similarities and differences between these techniques, and extracted common characteristics that we then use to group them into different classes. These classes can then aid in the development of a detection algorithm that eventually looks for the presence of such a technique in an arbitrary malware sample found in the wild.

Table 1 presents a summary of our findings. In the following sections we will go over the identified characteristics, as well as the rationale behind the classes that we extracted from these characteristics.

3.1 Common Characteristics

As mentioned before, one of the first steps in classifying code injection techniques is to identify characteristics that describe the general nature of the implemented technique. In the following, we will introduce these characteristics, and explain the meaning behind the columns in Table 1.

Moment of Execution. This trait describes the moment in which the code can be injected and executed in the victim process. Some techniques allow for injecting the payload at any point in time while the victim process is running, whereas in others it is only possible to inject the code upon startup of the victim process or the underlying operating system itself.

Transmitter. The transmitter is the process that is responsible for performing the transmission of the code. Some techniques require the injector to perform the injection themselves, whereas others make sure that the victim process is tricked into loading the malicious code instead.

Catalyst. The catalyst describes the process that is eventually responsible for triggering the execution of the final payload. Similar to the *Transmitter*, some techniques implement the activation on the injector's side, whereas others wait for the victim process to trigger execution on their own.

File Dependency. Some techniques require a physical copy of the injected code on the disk, usually in the form of a dynamically loaded library file (on Windows this is a file with the `.dll` extension). This often means that such a file needs to be dropped before execution can take place.

Process Model. This trait describes the way in which malware selects and interacts with the victim process. For example, some techniques interact with already running processes, while others spawn new ones. Alternatively, some do not interact with a process directly at all, and instead let the underlying operating system do its job.

Threading Model. Similar to *Process Model*, this trait describes the dependence on threads of the technique. Some techniques require the creation of new threads, while others depend on manipulating existing threads, or let the underlying operating system handle this instead.

Memory Manipulation Model. This characteristic describes the dependence on manipulating the memory space of the victim process directly. Techniques that implement a memory manipulation model require specific parts of the victim process being tampered with, or allocate new pages of memory instead. This trait often goes hand in hand with creating or opening a process first, and is present in most classic code injection techniques.

Shellcode Dependency. These techniques require a small chunk of code to be injected directly into the victim process to let the victim process execute the final payload. Injecting this shellcode often requires a direct memory manipulation.

Configuration Model. Some injection techniques depend on changing specific settings of the victim process or underlying operating system. Samples in this category may make changes to the Windows Registry, or install malicious plugins in a user application such as a web browser. Often, these techniques also rely on the existence of a file on the disk.

			Technique	Moment of Execution[1]	Transmitter[2]	Catalyst[2]	File Dependency	Shellcode Dependency	Process Model[3]	Threading Model[4]	Memory Manipulation Model[5]	Configuration Model
Active	Intrusive	Destructive	Process Hollowing [52]	P	I	I		✓	N	E	N	
Active	Intrusive	Destructive	Thread Execution Hijacking [53]	A	I	I		✓	E	E	N	
Active	Intrusive	Destructive	IAT Hooking [37]	A	I	V		✓	E		E	
Active	Intrusive	Destructive	CTray Hooking [51]	A	I	V		✓	E		E	
Active	Intrusive		APC Shell Injection [46]	A	I	V		✓	E	E	N	
Active	Intrusive		APC DLL Injection [46]	A	I	V	✓		E	E	N	
Active	Non-Intrusive		Shellcode Injection [32]	A	I	I		✓	E	N	N	
Active	Non-Intrusive		PE Injection [62]	A	I	I		✓	E	N	N	
Active	Non-Intrusive		Reflective DLL Injection [32]	A	I	I		✓	E	N	N	
Active	Non-Intrusive		Memory Module Injection [32]	A	I	I		✓	E	N	E	
Active	Non-Intrusive		Classic DLL Injection [26, 50]	A	I	I	✓		E	N	N	
Passive	Configuration		Shim Injection [33]	P	V	V	✓					✓
Passive	Configuration		Image File Execution Options (IFEO) [56]	L	V	V	✓					✓
Passive	Configuration		AppInit_DLLs Injection [48]	L	V	V	✓					✓
Passive	Configuration		AppCertDLLs Injection [47]	L	V	V	✓					✓
Passive	Configuration		COM Hijacking [23]	L	V	V	✓					✓
Passive			Windows Hook Injection [27]	A	V	I	✓					

[1] **A**: At any time, **P**: On Process Start, **L:** On Library Load.
[2] **I**: Injector Process, **V**: Victim Process.
[3] **N**: New Process Creation, **E**: Existing Process Manipulation.
[4] **N**: New Thread Creation, **E**: Existing Thread Manipulation.
[5] **N**: New Memory Allocation, **E**: Existing Memory Manipulation.

Table 1: Overview of code injection techniques and their characteristics.

3.2 Taxonomy

With the help of the identified common characteristics, we can move on to extracting different core characteristics that place different techniques into a set of groups. These groups are highlighted on the left hand side of Table 1, and subdivides the table using horizontal lines. In the following we will discuss the rationale behind these classes.

3.2.1 Active and Passive Injections

The most important distinguishing feature that we observed has to deal with the level of interaction that is required by the technique. For example, a large group of techniques either directly communicate with the victim process (by the means of opening process or thread handles) and either allocates new pages of memory, or manipulates existing pages instead. This is an important feature as it contributes to the stealthy capabilities of the technique. Since these kinds of interactions often translate to well known sequences of API calls, these can be observed more easily by monitoring software. Therefore, let us introduce the concept of *active* code injection techniques:

Definition 5 (Active Techniques) *A code injection technique is called an* ***active*** *injection if it requires direct interaction with the victim process or one of its threads, or actively makes changes in the victim process' memory.*

A lot of the existing techniques can be considered an active injection technique. For example, *Shellcode Injection* opens a handle to the victim process, and uses it to directly inject executable memory into it with the help of an API function such as `NtWriteVirtualMemory` [32]. However, a technique that abuses for example the shims infrastructure does not directly communicate with the target process, nor does it actively change its memory. Rather, it lets the underlying operating system load and execute the code instead [33]. This is much more of a *passive* approach, and therefore would not be classified as an active technique.

3.2.2 Intrusiveness and Destructiveness

We can further sub-categorize active techniques by looking at the type of interaction that is required. For example, some techniques interrupt and manipulate the original execution of the victim process. Sometimes this happens to such an extend, that parts of the application or the entire process completely stop working properly. If an application suddenly stops working or starts doing something noticeably different, then this can be picked up on relatively easy as well. Therefore, let us introduce the notion of intrusive and destructive injection techniques:

Definition 6 (Intrusiveness) *A code injection technique is called* ***intrusive*** *if (parts of) the victim process' memory or threads are changed.*

Definition 7 (Destructiveness) *A technique is called* ***destructive*** *if it is intrusive and (parts of) the application stop(s) working as a result of the intrusive intervention.*

An example of a destructive technique is *Process Hollowing*, which creates a new victim process in a suspended state, and replaces the memory contents with new code [52]. As a result, upon resuming, the victim process is not doing its original work anymore, which indicates the destructive behavior. This is in contrast with for example *Classic DLL injection*, which simply forces the target application to load a library on the disk without interrupting any thread [26]. Since it does not change any existing memory or thread context, this technique therefore falls under the *non-intrusive* category instead.

3.2.3 Configuration-based Injections

A final subdivision was made in the Passive code injection techniques. This subdivision groups techniques together that require specific changes in the registry to be made. This is a direct result of the *Configuration Model* trait, as these are the only techniques that have this characteristic. An example of such a technique is *AppInit_DLLs Injection*, which requires registering a library file into the Windows Registry. On the other hand, the *Windows Hook* injection technique interfaces with system events directly,

and does not require a persistent configuration stored on the disk.

4 Methodology

We now proceed with describing our methodology that we use to decide whether a sample implements code injection.

Since malware developers often obfuscate or pack their samples before they are released into the wild, static analysis is not a feasible solution. For this reason, we opt for an approach that is based on dynamic analysis instead. This means that our detection system will run a sample in an isolated sandbox, and record a stream of side-effects, which from now on we will be referring to as the *event stream*. For our purposes, we mainly focus on API calls and the arguments passed onto them, but it is important to note that the models that we introduce can easily be extended to *any* type of event that can be observed by the underlying sandboxing technology.

The task is to map patterns within the recorded event stream to the identified techniques. This is quite similar to recognizing patterns in a symbol or token stream, as is done by many different parsers and compilers for programming languages [22]. As such, we choose to use a similar approach.

In the following sections, we discuss how token streams slightly differ from our event streams in terms of quality and consistency of the data, and that this difference introduces a couple of challenges that need to be addressed. We do this by rewriting the problem into a similar problem that has been studied in the field of distributed systems. We then revisit the modelling language of Petri Nets that is used to describe these types of systems, and introduce an extension to this language which we call *Behavior Nets*. This extension allows us to overcome the challenges, and make it possible to model the traits as identified in the previous section.

4.1 Behavior Recognition as a Concurrent System

In this section, we discuss two main challenges that need to be overcome while recognizing patterns in a recorded event stream. We then show that the problem is equivalent to monitoring a concurrent system, where multiple threads performing operations in parallel can be seen as a single thread with a random interleaving of operations.

4.1.1 Noise and Reordering of Operations

One of the main challenges with event streams is that the raw data within an event stream is a lot more fuzzy than for example a source code file written in a programming language. Since we are monitoring an entire system, a lot more noise is present. Signals that are produced by other running processes or internal functions within the operating system itself, can clutter the input stream with a lot of extra data points that needs to be discarded.

Furthermore, the exact order in which the symbols appear in the stream is not always clear. This is especially the case when certain steps in an algorithm or procedure are independent of each other. For example, if an operation C depends on the execution of A and B, but A and B are completely independent of each other, then it does not matter whether first A or first B is executed. As long as both are finished before operation C is invoked, this does not cause any difference in the final effect of the program. This insight has proven to be very useful for malware developers to avoid detection. If an anti-virus only has a signature for the sequence A, B, C, then the malware can simply perform B, A, C instead to get to the same result while staying undetected.

One naive solution to this problem is to enumerate all possible orderings of a certain behavior, but this is very inefficient in space as it grows exponentially in the number of independent operations. Ideally, a system that does not depend on this raw sequencing of operations, but rather is able to detect the dependency relations between them, is much more robust against these types of mutations.

4.1.2 Reduction from Concurrent Systems

The key insight that we are going to use, is that recognizing behavior in a single event stream, where the order of independent operations does not matter but the general dependency does, is the same as recogniz-

ing behavior in a concurrent system where multiple independent processes run at the same time.

Consider three threads A, B and C, where A and B run concurrently and C waits for A and B to finish before it continues its execution. If we record all the events produced by the three running threads, and order them by time, we produce a new single event stream. This stream starts with a random interleaving of the two original event streams produced by A and B, and is followed by the event stream of C in its entirety.

Now consider another thread D, which performs the exact same operations of A, B and C in this exact same order. This scenario is analogous to the example as described in section 4.1.1. What emerges is a resulting stream that indistinguishable from the stream we constructed earlier from the threads A, B and C. This shows that modelling concurrent behavior can be reduced to modelling a single threaded system where independent operations might be ordered in a non-deterministic manner. We will use this result to build up models that can handle arbitrary rearrangements of independent steps.

4.2 Petri Nets

One of the ways to model concurrent systems is with the help of Petri Nets. Let us first recall the definition of a net:

Definition 8 (Net) *A net is a tuple $N = (P, T, F)$, where P and T are disjoint finite sets of nodes, representing* places *and* transitions *respectively, and $F \subseteq (P \times T) \cup (T \times P)$ denotes the set of arcs, such that together they form a bipartite graph.*

Petri nets are nets where places may contain any number of marks called *tokens*. Furthermore, arcs between the places and transitions are weighted [54]. More formally:

Definition 9 (Petri Net) *A Petri net is a tuple $PN = (N, M, W)$, where $N = (P, T, F)$ is a net, $M : P \to \mathbb{N}$ a function that assigns a number of tokens to every place, and $W : F \to \mathbb{N}$ a function that assigns a weight to every arc.*

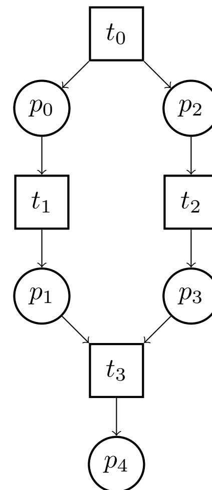

Figure 2: An example Petri net with a fork-join construction.

A transition is said to be *enabled* if there are enough tokens on every input place according to the weight of the incoming arc (for each input place s: $M(s) \geq W(s, t)$). When a transition is enabled, it can be *fired*. If this happens, the amount of tokens as indicated by the weights of the arcs are consumed from all the input places, and new tokens are produced at all the output places. This last remark means that if a transition is a branch with two or more output places (such as t_0 in Figure 2), it does not encode a choice as is the case with other types of state machines or control flow graphs. Rather, it can be compared to a *fork* where multiple processes start running concurrently. Conversely, two converging edges (such as the ones at t_3) can be used to model a *join* of multiple threads, where two processes wait for each other to complete.

Important to note here is that the order in which the transitions are fired can be completely non-deterministic, as is the case with concurrent systems.

4.3 Behavior Nets

While Petri nets can model concurrent behavior, they do not place any semantics on tokens and transitions. Transitions can fire at any time as long as there are

enough tokens in its input places. For our purposes, we will therefore extend the concept of a Petri net, and introduce *Behavior Nets*. The main idea is to map events observed in the system (e.g. a call to an API function) to the transitions in the net. These transitions will then only be enabled and fired if there are enough tokens in its preset, *and* match a certain predicate on the event.

4.3.1 Definitions

A behavior net works on a set of symbolic variables for which concrete values are found as events are consumed from the event stream. These symbolic variables are not part of the original program that is being observed, but rather are variables that solely exist within the detector alone. A token in a behavior net represents one concretization of such a set of symbolic variables, and can be seen as a (partial) mapping between symbolic variables and their concrete values. Two tokens can be combined together. The result is a new mapping that uses the values of both original mappings. If there exists a symbolic variable α which is assigned two different values in both original tokens, we speak of tokens that are in *conflict*. Combining conflicting tokens results in $\bot$, the invalid token. Combining any other token with $\bot$ will also result in $\bot$.

We add to every transition t in the net a corresponding transition function δ_t. This function takes one recorded event from the observed system, as well as an input token. The idea is that δ_t transforms the input token into a new token if and only if the input event and token match the expected pattern, and otherwise returns $\bot$.

More formally, let $\mathcal{S}$ be the set of all symbolic variables, $\mathcal{Z}$ be the set of all possible values that every $s \in \mathcal{S}$ can be assigned with, $\mathcal{T} = \mathcal{P}(\mathcal{S} \times \mathcal{Z})$ be the set of all tokens, and Σ be the set of all possible events that can happen. Then we can define a behavior net as follows:

Definition 10 (Behavior Net) *A behavior net is a tuple* $BN = (N, A, M, \delta)$*, where*

- $N = (P, T, F)$ *is a net,*
- $A \subseteq P$ *is a set containing the accepting places,*
- $M : P \rightarrow \mathcal{P}(\mathcal{T})$ *is a function that assigns a set of tokens to every place in the net, and*
- $\delta : T \rightarrow (\Sigma \times \mathcal{T} \rightharpoonup \mathcal{T})$ *is a function that assigns partial transition functions to every transition in the net.*

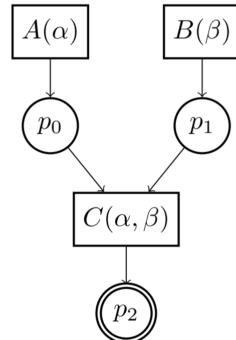

Figure 3: Sample behavior net where the event C depends on the arguments α and β. These two values are to be observed from two independent events A and B. p_2 is an accepting state, which is indicated by the double outline.

Once there exists at least one token in any of the accepting places, the behavior is considered *recognized*. Figure 3 depicts an example of a behavior net that is able to recognize the example pattern given in Section 4.1.2.

The execution semantics of a behavior net are very similar to a normal Petri net, with only a few changes. We will discuss these differences in the following sections.

4.3.2 The Transition Functions

As with normal Petri nets, a transition t is enabled only when there are enough tokens at its input places. However, in contrast to normal Petri nets, all possible combinations of tokens are considered at once. Each combination has the potential to produce a new token, depending on the implementation of δ_t. This way we do not really have a concept of *weighted arcs*, and as such this is not included in the definition. This also means that it is possible for a transition to produce multiple tokens at the same output place.

The reason why we consider all possible combinations of tokens at once, is because upon firing a transition we do not know yet which combination of input tokens is a combination that will eventually lead to a token in an accepting place. Choosing only a single token arbitrarily at once might therefore result in choosing the wrong token and making the behavior net get stuck and not progress further.

Algorithm 1 describes the process of determining the new tokens when a transition is fired. Every combination of input tokens is combined into a single token. This new token is then fed into the corresponding δ_t together with the current event to process. If it returns $\perp$, then this new token is discarded. Otherwise, it is added to the result and thus will be propagated to every output place of the transition. In the case that there are no input places, the empty token is provided to δ_t, and only one token is produced instead.

Algorithm 1 Enumerate new tokens for transition t on event e.

```
1:  procedure EnumNewTokens(t, e)
2:      n ← |input places of t|
3:      if n = 0 then
4:          r ← {δ_t(e, ∅)}
5:      else
6:          r ← ∅
7:          Q ← {M(p)|p ∈ input places of t}
8:          for all (x_1, .., x_n) ∈ Combinations(Q) do
9:              x ← TokenCombine(x_1, .., x_n)
10:             if δ_t(e, x) ≠ ⊥ then
11:                 r ← r ∪ {δ_t(e, x)}
12:     return r
```

This setup allows δ_t to decide whether a certain observation is part of a chain of events that we are interested in, and not background noise that was introduced by other processes. For example, suppose δ_t matches on calls to the Windows API function `NtCreateThreadEx`, which allows for creating threads in any running process. Without also using an input token in our matching criteria, δ_t would only be able to match on *any* instance of this event. This makes the event indistinguishable from other calls to the same function (see Table 2 and Table 3 for example traces). Since `NtCreateThreadEx` is a very commonly used function, this results in a high potential for false positives to arise. However, if δ_t were to also consider the arguments that were used to call the function, we can e.g. verify that the first argument (the process handle) matches an argument that was observed in prior events such as function calls to `NtAllocateVirtualMemory` or `NtWriteVirtualMemory` (responsible for allocating and injecting the executable code respectively). By letting transition functions assign concrete values to symbolic variables in a token, they can communicate these values to other transitions in the net. This way, a behavior net can decide with more confidence which events are related to each other, and which can be filtered out. An example of a behavior net that implements this, is given in Figure 4.

Time	Observed event
...	...
t	`NtAllocateVirtualMemory(0xA0, ...)`
$t+1$	`NtWriteVirtualMemory(0xA0, ...)`
$t+2$	`NtCreateThreadEx(0xA0, ...)`
...	...

Table 2: An excerpt of an events stream, recorded from a system running a sample applying the Shellcode Injection technique.

Time	Observed event
...	...
t	`NtAllocateVirtualMemory(0xA0, ...)`
$t+1$	`NtWriteVirtualMemory(0x42, ...)`
$t+2$	`NtCreateThreadEx(0xB8, ...)`
...	...

Table 3: An excerpt of an event stream, recorded from a system with processes running similar functions as used in Shellcode Injection, but are unrelated to each other.

4.3.3 Token Consumption

The second difference with Petri nets is that tokens are no longer removed upon transitioning. Once a

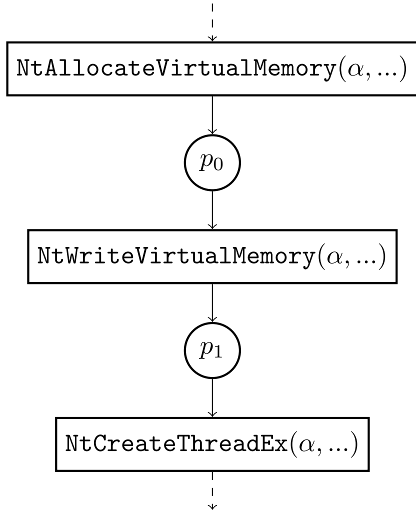

Figure 4: An excerpt of a behavior net that captures the Shellcode Injection technique. It uses a single symbolic variable α to link three events together based on the first argument of the events.

token is produced and put in a place, it will always remain in that place and never be destroyed. The reason behind this, is that it allows for backtracking without introducing any extra logic. For example, consider a model such as the one in Figure 5, and a sequence of events which contains the subsequence $(f(x), g(y), g(z), h(z))$. Clearly, if we set $\alpha = x$ and $\beta = z$, then this would match the pattern $(f(\alpha), g(\beta), h(\beta))$ as indicated by the net. Yet with the default execution rules of a Petri net, this would not be recognized. This is demonstrated in Table 4. Upon processing the first call to g, it would greedily consume the token stored at p_0, and the newly produced token at p_1 will set $\beta = y$. The problem is that upon processing the second g call, the transition between p_0 and p_1 would no longer be enabled, since no token is present any more at p_0. This causes the model to get stuck with a token that (incorrectly) assigns y to β, and $\beta = z$ will never be considered as an option. For this reason, tokens are preserved in a behavior net. Preserving the token at p_0 will ensure that the second g call will also be considered as an option, and as such the model can continue progressing. This is demonstrated in Table 5.

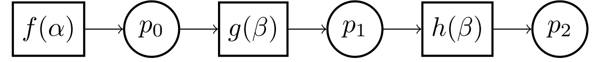

Figure 5: A behavior net with three transitions matching on different events f, g and h. The last two transitions share a symbolic variable β, indicating the arguments for both g and h need to be the same value.

A clear downside of this is that not consuming tokens can potentially result in *overflowing* the net with tokens. However, since our models are relatively small and do not contain cycles, this is only a theoretical issue that would not be a problem in practice. Furthermore, we discard any duplicated tokens present at a single place. This is an acceptable change, since a duplicated token does not provide any extra information about a potential final matching of symbolic variables to their concrete values.

e	$M(p_0)$	$M(p_1)$	$M(p_2)$
$f(x)$	$\{\alpha = x\}$		
$g(y)$		$\{\alpha = x, \beta = y\}$	
$g(z)$		$\{\alpha = x, \beta = y\}$	
$h(z)$		$\{\alpha = x, \beta = y\}$	

Table 4: The evolution of the marking of Figure 5 with token consumption.

e	$M(p_0)$	$M(p_1)$	$M(p_2)$
$f(x)$	$\{\alpha = x\}$		
$g(y)$	$\{\alpha = x\}$	$\{\alpha = x, \beta = y\}$	
$g(z)$	$\{\alpha = x\}$	$\{\alpha = x, \beta = y\}$, $\{\alpha = x, \beta = z\}$	
$h(z)$	$\{\alpha = x\}$	$\{\alpha = x, \beta = y\}$, $\{\alpha = x, \beta = z\}$	$\{\alpha = x, \beta = z\}$

Table 5: The evolution of the marking of Figure 5 without token consumption.

4.4 Modelling Code Injection using Behavior Nets

We now continue by expressing the code injection techniques as described in section 3.1 into behavior nets. We do this by looking at the core characteristics of each technique to build up the individual transitions, and connect them in such a way that it follows the general pattern of the behavior.

Figures 6 to 14 depict all the nets that we built. In these nets, we can see the expression of certain traits in the form of API calls. For example, in the Classic DLL injection technique, threads are created after a file was dropped. This is reflected by the transitions matching on specific calls to `NtCreateFile`, `CreateRemoteThread` and `NtCreateThreadEx` in Figure 9. Furthermore, in Figures 6 and 12 we can see that in both the APC Shell Injection and the Process Hollowing technique, threads are manipulated instead. This is reflected by the transitions matching on calls to `NtQueueApcThread`, `NtSetContextThread` and `NtResumeThread`. Process Hollowing however, creates a new process which is expressed using the `NtCreateUserProcess` transition. This is in contrast to manipulating an existing one, as is the case with the former technique. As such, the former matches on the `NtOpenProcess` system call instead.

We can also see that some transition nodes make use of a transition function that puts extra constraints on the captured symbolic variables. For example, in Figure 6 we can see that the node matching on `NtWriteVirtualMemory` restricts the value of β to the interval $\{\alpha, ..., \alpha + \sigma\}$. This range is inferred from a previous transition node matching on `NtAllocateVirtualMemory`, indicating that β should be a memory address that falls within a previously allocated address range.

Notice also that the behavior nets for configuration based techniques can all be summarized using Figure 14, and are very small compared to the ones for active techniques. This stems from the fact that these techniques only require one change in the registry for them to achieve both a transmission as well as a catalyst. After making the change in the registry, the underlying operating system will always do the remainder of the heavy lifting afterwards automatically. This means that we do not have to add any additional transition nodes to decide that code injection will happen, and as such, these nodes can be omitted from the net.

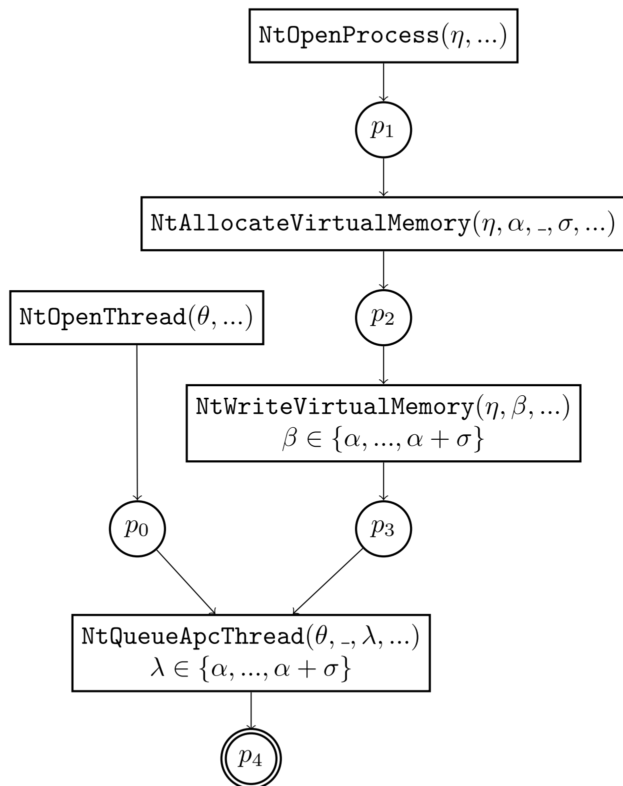

Figure 6: A behavior net modelling the *APC Shell Injection* technique. Since not all parameters in every function call are necessary to match on, we use an underscore ('_') and ellipses ('...') to indicate they are discarded.

5 System Architecture

We now move on to the design of our system that uses the concepts of Behavior nets to automatically recognize the use of code injection in a given sample. Figure 15 depicts an overview of the system that we

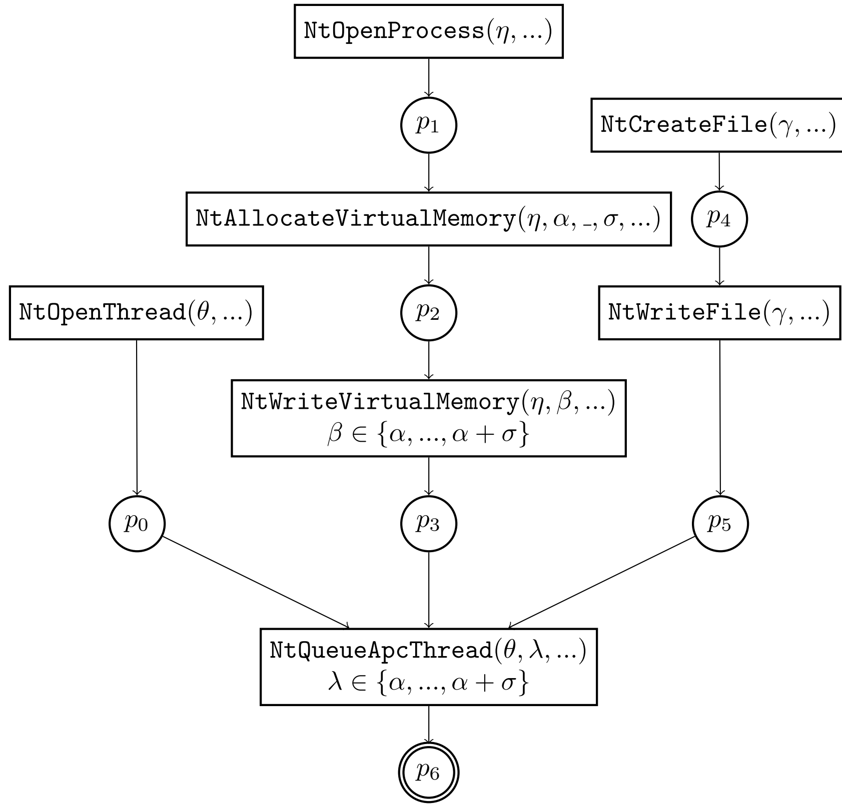

Figure 7: A behavior net modelling the *APC DLL Injection* technique. This is similar to Figure 6, but contains an additional branch ensuring that a file drop is registered before the call to `NtQueueApcThread`. Also the pattern matching for this last function call is slightly different: λ matches on the second parameter, rather than the third parameter of this function.

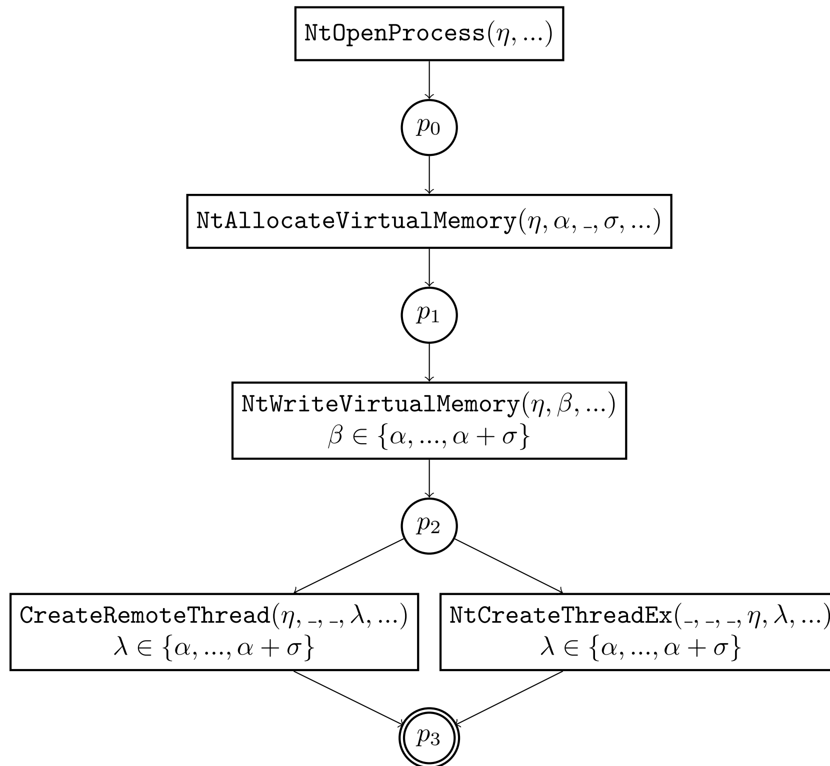

Figure 8: A behavior net modelling the *Generic Shellcode Injection* technique. In this graph we can see that p_2 branches into two possible function calls. This is because the injector process may trigger execution in the victim process using either of these two API functions.

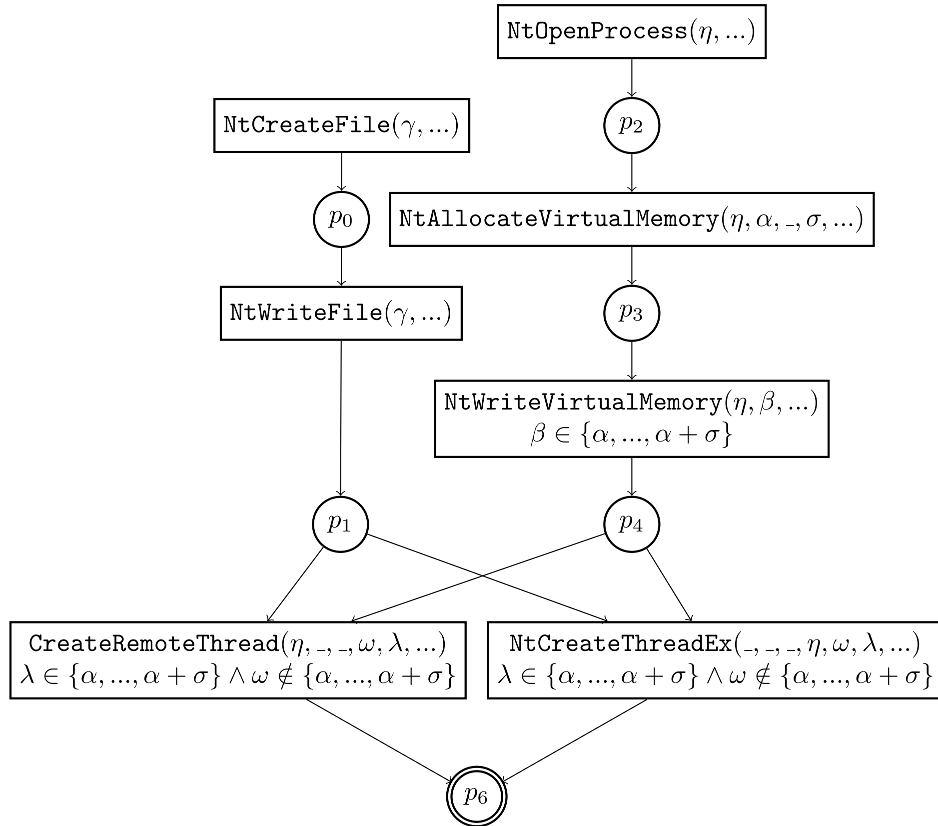

Figure 9: A behavior net modelling the *Classic DLL Injection* technique. Similar to Figure 7, this also contains an additional branch that checks for a dropped file. Furthermore, we see a similar branching construction as in Figure 8, indicating one of two possible system calls may be used in the end.

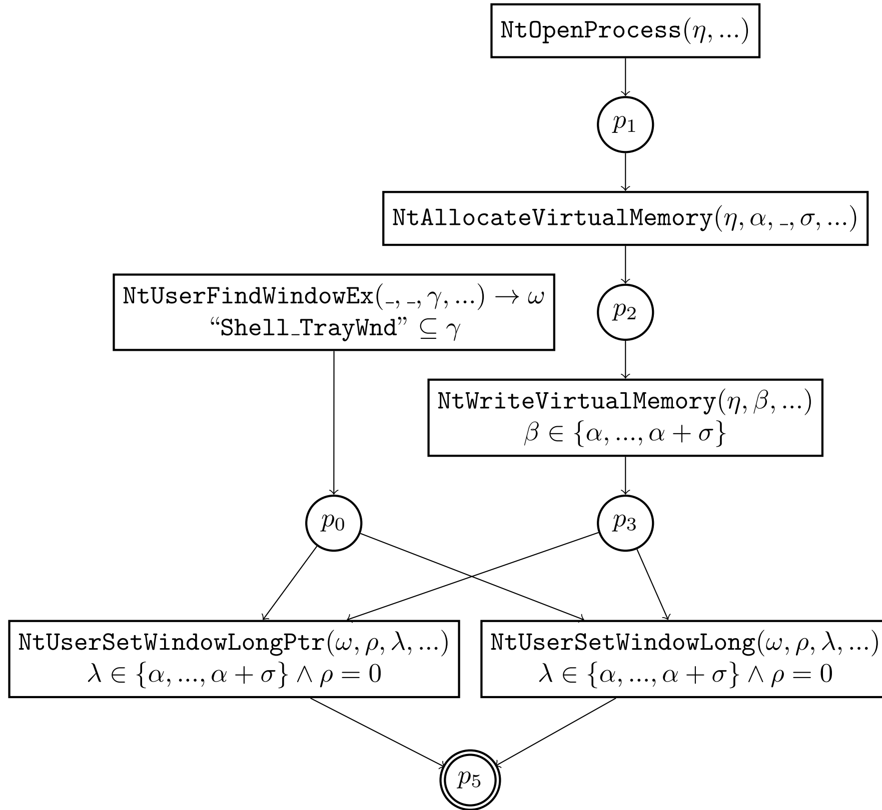

Figure 10: A behavior net modelling the *CTray VTable Injection* technique. In this graph, we use the $\rightarrow$ operator to indicate the return value of the function `NtUserFindWindowEx` is captured by the ω symbolic variable.

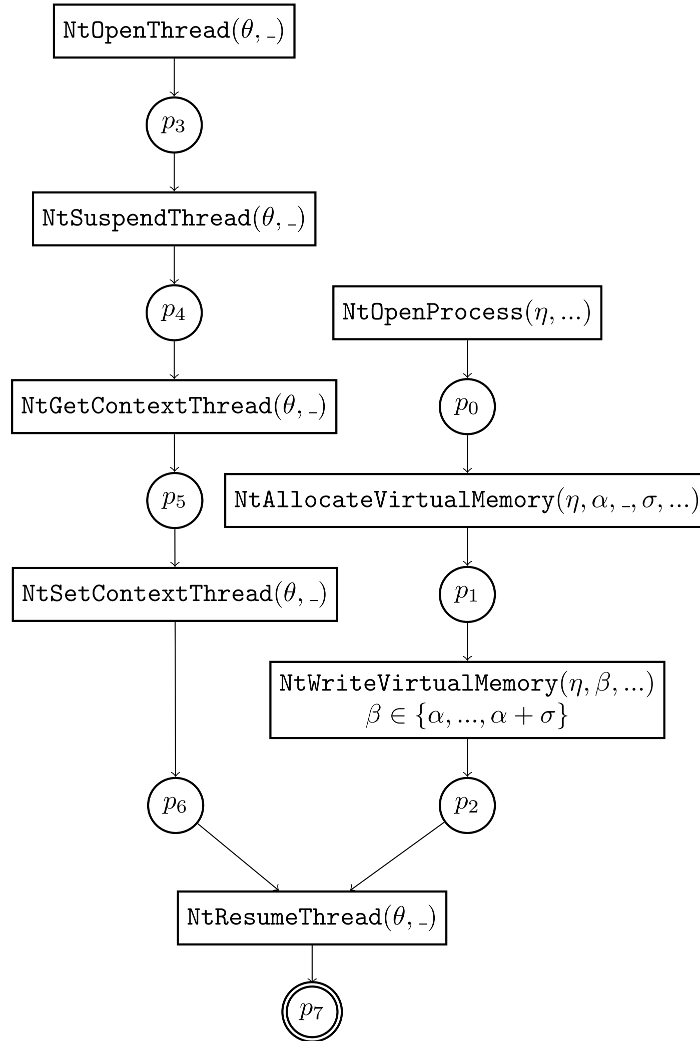

Figure 11: A behavior net modelling the *Thread Execution Hijacking* technique.

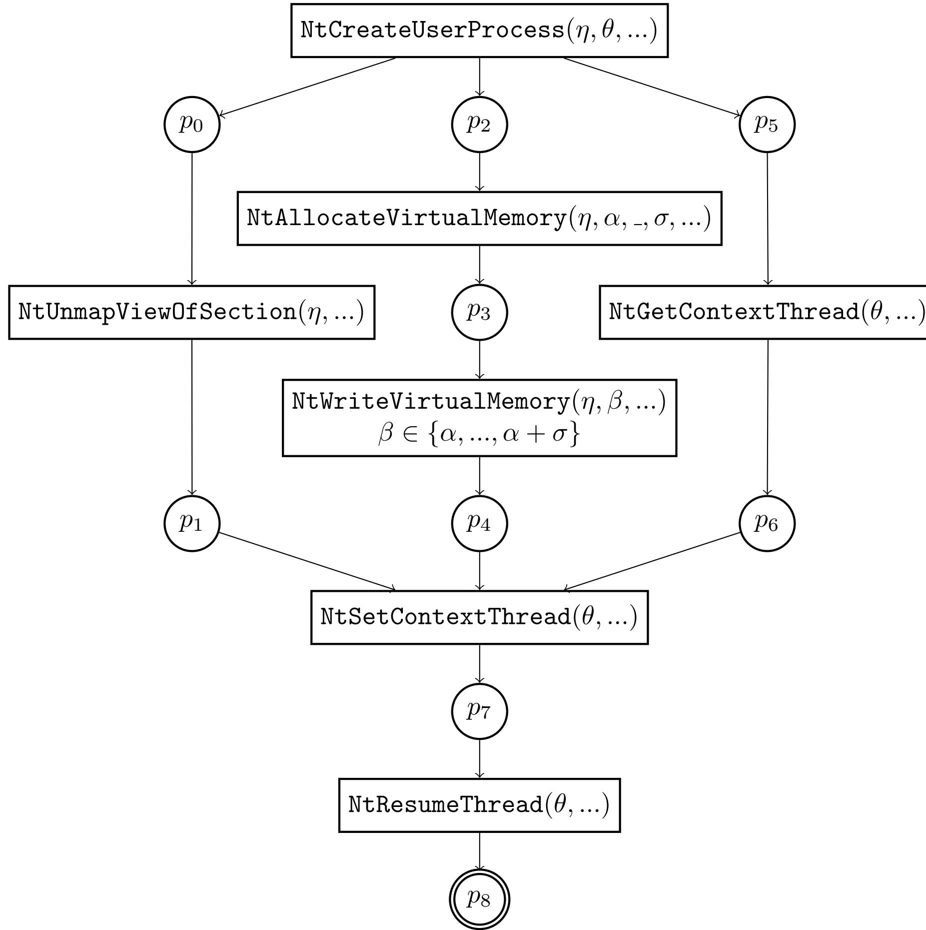

Figure 12: A behavior net modelling the *Process Hollowing* technique. Here we see the first transition branching into three different places. This indicates that after the call to `NtCreateUserProcess`, three different tasks might be executed in any order or might be interleaved into each other. However, all three branches converge into the same node matching on the `NtSetContextThread` function. This indicates that all three tasks must complete before this system call is made.

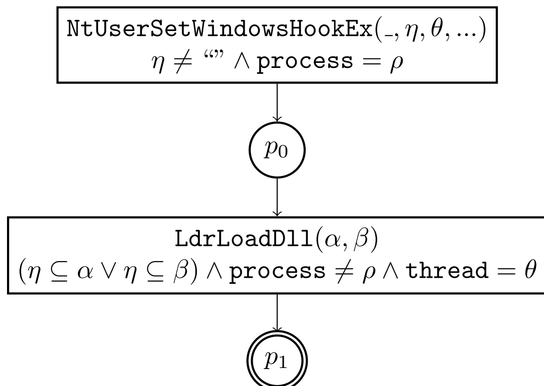

Figure 13: A behavior net modelling the *Windows Hook* technique. In this graph, we indicate the process and thread that are responsible for producing the event with `process` and `thread`.

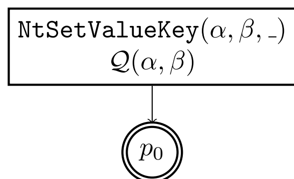

Figure 14: The general set up for a behavior net that models a configuration based code injection technique. In this small graph, $\mathcal{Q}$ refers to a predicate that tests whether the right registry key is accessed for this technique, according to the first and second arguments of the observed `NtSetValueKey` system call.

developed. It consists of two components; the *Detector* and the *Examination Environment.* The Detector acts as a front-end for the system, taking samples as input, and reporting back the final verdict. It does so by uploading the samples to the Examination Environment, which runs them for a limited amount of time in a isolated sandbox. During the execution, an event stream of the sandbox is recorded. After the examination has completed, the Detector then downloads this event stream, and runs it through a set of Behavior nets that model the different types of techniques as identified in Section 3. Finally, based on the final markings of these Behavior nets, it will then compile a detection report that includes all the techniques that were fully recognized.

In the following, we will discuss how each component works in more detail.

5.1 The Detector

As mentioned before, the Detector is the main driving force for uploading samples and analyzing event streams. The input of the detector is the path to a single sample, or a directory containing multiple samples. All samples that need to be analyzed are added to a queue (in the figure displayed as the Task Queue), and are uploaded one by one to the examination environment.

The Detector is also responsible for maintaining all Behavior nets that need to be considered while analyzing the resulting event streams. Important to note is that these Behavior nets are obtained from a repository of Behavior net specifications that reside on the disk. For this, we built a small Domain Specific Language (DSL) that is inspired by the DOT graph modelling language [7], as well as Haskell that features pattern matching syntax [4]. This makes the detector easily extensible, should in the future new types of techniques be discovered, or other types of behaviors need to be detected.

In our DSL, we define Behavior nets with a `behavior` block. Within this block, we introduce the places, the transitions and arcs between them.

```
behavior "Name" {
   ...
}
```

Places are defined using the `place` keyword, followed by one or more identifiers. If such a declaration ends with the word `accepting`, all the places within that declaration will be included in the accepting places. For example, the declaration for `p5` below is marked as an accepting place, while the places `p0` to `p4` are not.

```
place [p0 p1 p2 p3 p4]
place p5 accepting
```

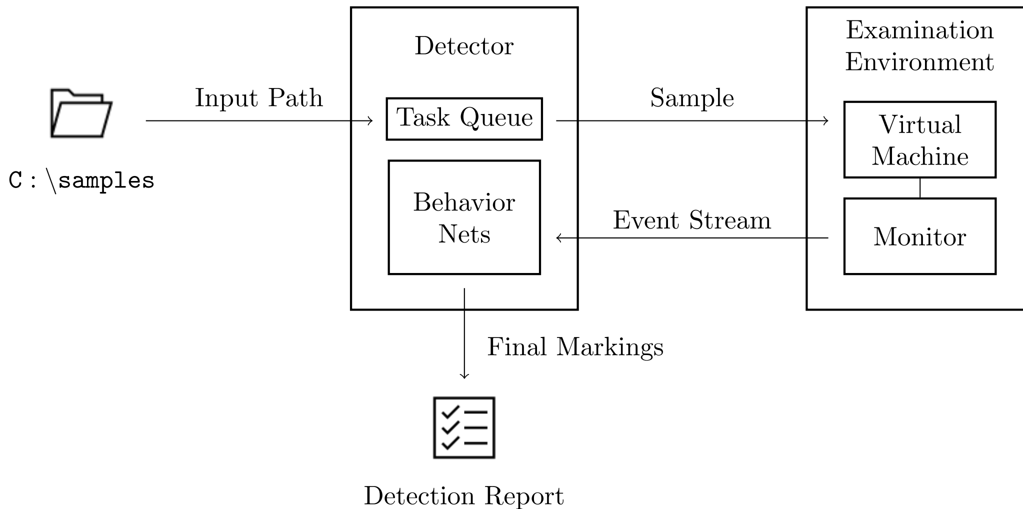

Figure 15: System Architecture Overview

Transitions in a behavior net are defined using `transition` blocks. Inside a transition block, the transition function δ_t is configured. This starts off by indicating the name of the event to match, followed by a set of symbolic variables that capture the arguments of the event. If an argument is not relevant to the detection of the behavior, we use an underscore (_) to discard it. Extra constraints can also be added to these symbolic variables by including a `where` clause. An example of such a constraint can be seen in transition `t2` below, where we restrict the value of `x` to be an address within a chunk of memory allocated by a prior call to `NtAllocateVirtualMemory` in transition `t1`.

```
transition t1 {
  NtAllocateVirtualMemory(h, a, _, s, _, _)
}

transition t2 {
  NtWriteVirtualMemory(h, x, _, _, _)
  where
    x in [a .. (a + s)]
}
```

Additionally, symbolic variables can be defined for the processes and threads that were responsible for producing the event. This is done using an `in` clause within the block, and is in particular useful for encoding that the catalyst should be the victim process, as is the case with *Windows Hook Injection*. An example of such an `in` clause is shown in the following code snippet.

```
transition t1 {
  NtUserSetWindowsHookEx(
     _, cbDll, tid1, _, _, _ , _)
  in
    process pid1
}

transition t2 {
  LdrLoadDll(name, path)
  in
    process pid2
    thread tid1
  where
    (cbDll in name) or (cbDll in path)
    pid2 != pid1
}
```

Similar to the DOT language, we use the `->` operator to add arcs between transitions and places. It is also possible to chain multiple nodes together in the same line by using multiple `->` operators in sequence.

```
t0 -> p0
t1 -> p1 -> t2 -> p3 -> t3
```

A full example of a Behavior net expressed in our DSL can be found in Listing 1 in the appendix.

5.2 The Examination Environment

To be able to perform dynamic analysis, we rely on running samples in an isolated execution environment. For a sandboxing solution, we chose the DRAKVUF Sandbox as our main driver [41]. DRAKVUF is a black-box malware analysis system that is able to run arbitrary samples inside a Virtual Machine (VM) for a limited amount of time. Like other sandboxing solutions, it is able to monitor for activity such as API calls, system calls, network traffic and file system events. All these events are recorded into a set of log files, which can be downloaded and processed by our Detector program after the examination has completed.

One main advantage of DRAKVUF in comparison to other solutions, is that it is able to monitor an entire system as opposed to just single processes. Given the nature of code injection techniques, this is a feature that greatly increases the probability of our detector to observe the expected behavior. Furthermore, DRAKVUF observes the activity from outside of the VM by interfacing directly into the underlying virtualization software. This means that it does not require an agent within the VM itself to do the instrumentation and monitoring (as opposed to solutions such as Cuckoo Sandbox [2]). Consequently, this vastly reduces the risk of being fingerprinted by a sample, and thus increases the potential for the sample to actually activate itself.

We use Windows 10 as an operating system for the VM, since this is the most market dominant operating system by the time of conducting this research [3]. To remove any potential interference from other programs, we disable various background services such as the Windows Search Indexer, Windows Update, User Account Control (UAC) and Windows Defender. This is important, as these services might unnecessarily increase the size of the resulting event streams, prevent the malware sample from running, or introduce artifacts in the streams as a result of their own use of code injection techniques to do their own monitoring.

We do provide the malware with access to the internet, since some malware families rely on an active connection with a remote control server, or use the availability of internet (or lack thereof) as a means of detecting whether it is running in a sandbox or not [58]. However, we make sure that this internet access is limited. All traffic goes through a set of strict firewall rules, where we block several well-known ports used by commonly used TCP and UDP based protocols such as SMTP and SMB. Additionally, we make sure that all packets whose destination is within the IP subnet of the university are dropped, avoiding any potential denial of service or spreading into the network that our test machine was put in.

Finally, after every examination, we roll back the VM to a clean snapshot to revert any side-effects that might have been introduced by the malware. This also stops any potential denial of service that still managed to slip past our defenses.

These countermeasures were verified by our supervisors and were approved by the Ethics Committee of the University of Twente.

6 Evaluation

For evaluation, next to unit testing our implementation, we conducted two experiments to collect empirical evidence that show that our system is functioning properly. The first experiment is a small scale experiment that aims to verify whether our system is sufficiently equipped for detecting the presence of code injection in a single sample. The second experiment is a larger scale experiment where we look at a large data set of real world malware samples, and look at the general prevalence of code injection, as well as the distribution of the different techniques. In the following sections, we will discuss the overall parameters of our system that we used, as well as the samples that we considered. We then continue by presenting our findings.

6.1 Sample Selection

As mentioned before, to verify that our models are a correct representation of the studied injection techniques, and can be used in an examination on a larger

scale, we first compiled a small set of samples for which we know the implemented technique. These samples are a collection of the custom made implementations that we made ourselves as described in Section 3.1, as well as handpicked real-world samples for every technique. The handpicked samples are included to verify that our behavior nets are not biased towards our own implementations, and are generic enough to also recognize the ones that we would find in the wild.

For our prevalence assessment, we used a random subset from the VirusTotal Academic Data Set [8]. To prevent overrepresentation of a single malware family, we used AVClass [61] to classify each sample by family, and limited the number of samples per family to 20. By this process, we selected a total amount of 3075 samples originating from 2017, 2019 and 2020, and ran them through the system over the course of 5 weeks. Finally, to test whether the assessments made by our detector are consistent, we picked several samples from our selection at random, and manually verified that these indeed implement the injection technique as determined by our detector.

6.2 Time Limit per Sample

As was discussed in section 2.2.2, an important decision to make for the detector is the amount of execution time that should be allocated for every sample. While theoretically it is possible that a sample might delay execution for a very long time, Kuchler et al. showed that around 65% of all malware samples requires less than 2 minutes to run till completion, and 81% will not need longer than 10 minutes [39]. Since code injection is very likely to be one of the first steps that a sample might perform, we assume that any step executed after the 10 minute mark is very unlikely to contain anything relevant. We can then calculate a rough expected run time required for a single sample to perform some form of injection as $E(t) \approx 0.65 \cdot 2 + 0.45 \cdot 10 = 5.8$ minutes. If we then take an estimated execution time slowdown of 5% as a result of the extensive logging, we deem 6 minutes an acceptable execution time limit per sample.

Technique	**Match**	**Exact**
Process Hollowing	✓	✓
Thread Execution Hijacking	✓	✓
IAT Hooking		
CTray Hooking	✓	✓
APC Shell Injection	✓	✓
APC DLL Injection	✓	✓
Shellcode Injection	✓	✓
PE Injection	✓	
Reflective DLL Injection	✓	
Memory Module Injection	✓	
Classic DLL Injection	✓	✓
Shim Injection	✓	✓
Image File Execution Options	✓	✓
AppInit_DLLs Injection	✓	✓
AppCertDLLs Injection	✓	✓
COM Hijacking	✓	✓
Windows Hook Injection	✓	✓

Table 6: Overview of all recognized code injection techniques. The *Match* column indicates some form of code injection was recognized. The *Exact* column indicates the system was able to properly identify the technique as well.

6.3 Results

In the following, we present the effectiveness of behavior nets for identifying the different techniques, as well as our findings of the prevalence of code injection in the general malware scene.

6.3.1 Detection Capabilities

Table 6 shows an overview of the detection capabilities of our system on our handcrafted sample set. In this table, we make a distinction between picking up on the presence of code injection, and exact identifications of the technique.

We can see that the use of virtually any of these techniques will result in the system noticing that *some* form of injection has happened, with the exception of *IAT Hooking*. In this technique, a “normal” API function that is imported by the victim process is reused as a catalyst for a piece of shell-

code injected by the malware. The malware places a *jump* instruction at the start of this function, and reroutes the control flow to the entry point of the injected code. This means that once the victim process invokes the function, the shellcode is activated instead. Even though this technique is classified as destructive, and has a very high chance of breaking the victim process in the end, we cannot recognize these types of injections using our system. This stems from the fact that behavior nets can only test for the *presence* of events. They are not able to pick up on faulty or crashing behavior, nor are they able to recognize the *absence* of an event as a result of rerouting a function. Furthermore, this technique only requires two calls to the function `NtWriteVirtualMemory` for transmitting and preparing the catalyst respectively. While we can observe calls to this function, DRAKVUF does not provide us with enough information in the event stream to distinguish between `NtWriteVirtualMemory` calls that place hooks or inject other types of memory instead. This makes it virtually impossible to recognize this kind of injection without changing the implementation of DRAKVUF itself.

Important to note is that this does not mean that destructive techniques cannot be observed or identified at all. Instead, the other three destructive techniques that were considered can be recognized properly. This is due to the fact that in these techniques the presence of the catalyst is much more apparent. For example, in the *Process Hollowing* technique, the catalyst always results in making calls to the `NtSetContextThread` and `NtResumeThread` functions. Unlike the case of *IAT Hooking*, these events have very clear arguments that can be traced back to previously observed events produced by the transmitter. This allows us to distinguish these events from the general noise of system calls more easily than in the case of the `NtWriteVirtualMemory` calls in the *IAT Hooking* technique.

We can also see in the table that for the three techniques *PE Injection*, *Reflective DLL Injection* and *Memory Module Injection*, our system can recognize the presence of an injection, but not exactly identify which technique was used. This is a result of a similar issue to the one mentioned in the above. Due to limited granularity of the observed event stream, some techniques will have a very similar if not identical pattern of events for their transmitters and catalysts. This is the case for these three techniques. With the current level of detail that DRAKVUF can provide us, these three techniques become indistinguishable from *Shellcode Injection*, and can therefore only be classified as such. However, this is a reasonable compromise. Since the only difference between these techniques is in the actual contents of the injected memory, they can be seen as a special case of shell injected code. Therefore, while classifying it as such does not completely reflect the exact behavior that is exhibited, it is not necessarily an incorrect classification either.

6.3.2 Prevalence Statistics

Table 7 summarizes the general observed prevalence of code injection within the sample set. A total of 343 samples (11.15%) was found to perform at least one type of code injection within the first 6 minutes of execution. Additionally, Table 8 and Figure 16 present the distribution of the different techniques that were implemented within these 343 samples. From these statistics, we can see that the technique *Process Hollowing* is with a total share of 41.69% convincingly the most popular choice among malware developers. We can also see that some techniques (such as *CTray* and *AppCertDLLs Injection*) are not observed at all. Table 9 condenses these findings by aggregating all techniques that fall into the same class into a single row. Here we can see that 61.52% of all samples perform an active injection, and 48.10% perform a passive injection. Important to note here is that these percentages do not add up to 100%. This is a result of some samples implementing multiple code injection techniques.

Since malware samples belonging to the same family often employ the same type of behavior [12], and families differ in size, some techniques might still be overrepresented in Figure 16, despite setting an upper limit of 20 variations per family. Therefore, Figure 17 presents a slightly different view on the data, where all samples within the same malware family are considered as one instead. In this graph, if at least one

	2017		2019		2020		Total	
Positive Samples	121	12.06%	135	13.24%	87	9.22%	343	11.15%
Negative Samples	882	87.94%	885	86.76%	965	91.73%	2732	88.85%

Table 7: Observed general prevalence of code injection in the sample sets from 2017, 2019 and 2020.

	2017		2019		2020		Total	
Process Hollowing	61	50.41%	55	40.74%	27	27.84%	143	41.69%
Thread Execution Hijacking	2	1.65%	0	0.00%	0	0.00%	2	0.58%
CTray Injection	0	0.00%	0	0.00%	0	0.00%	0	0.00%
APC Shell Injection	0	0.00%	0	0.00%	1	1.03%	1	0.29%
APC DLL Injection	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Generic Shellcode Injection	16	13.22%	13	9.63%	11	11.34%	43	12.54%
Classic DLL Injection	4	3.31%	18	13.33%	2	2.06%	22	6.41%
Application Shim Injection	0	0.00%	0	0.00%	0	0.00%	0	0.00%
Image File Execution Options	6	4.96%	12	0.000%	20	20.62%	38	11.08%
AppInit_DLLs Injection	17	14.05%	18	8.89%	20	20.62%	55	16.03%
AppCertDLLs Injection	0	0.00%	0	13.33%	0	0.00%	0	0.00%
COM Hijacking	20	16.53%	28	20.74%	11	11.34%	60	17.49%
SetWindowsHookEx Injection	6	4.96%	3	2.22%	14	14.43%	12	3.50%

Table 8: Observed distribution of code injection techniques implemented by malware in the sample sets from 2017, 2019 and 2020.

sample within a family was found to be performing a given type of code injection, then this family is considered to implement this technique as well. We can see that the general trends remain very similar to the previously found statistics. Process Hollowing is still the most prevalent, and is still followed by the collective of all passive injection techniques. Some notable differences are an increase in popularity for Generic Shellcode Injection and COM Hijacking, whereas techniques such as Image File Execution Options and AppInit_DLLs Injection dropped slightly.

7 Discussion

In this section, we discuss our findings by providing an additional perspective on the general observed prevalence, as well as the distributions of the techniques that were used by the examined samples.

7.1 General Prevalence

From Table 7 we can see that 11.15% of all observed samples uses some type of code injection. It is important to note is that this percentage is very likely to be an underestimation of the actual number of samples that use code injection. This has a couple of possible explanations, which we discuss in the following.

Firstly, while behavior nets are relatively generic models, they still are a form of signature-based detection. This means that if any of the tested malware samples used an unknown technique or a technique that is not captured by our models, then our system would not be able to detect it. This might also mean that for example the *IAT hooking* technique is used a lot, but remained unnoticed since we cannot express this technique in the form of a behavior net as was explained in Section 6.3.1.

Additionally, despite the counter measures in place, a sample could still not be activating itself dur-

	2017		2019		2020		Total	
Active	85	70.25%	86	63.70%	40	45.98%	211	61.52%
Intrusive	63	52.07%	55	40.74%	28	32.18%	146	42.57%
Destructive	63	52.07%	55	40.74%	27	31.03%	145	42.27%
Non-Intrusive	22	18.18%	31	22.96%	12	13.79%	65	18.95%
Passive	49	40.50%	61	45.19%	55	63.22%	165	48.10%
Configuration-Based	43	35.54%	58	42.96%	52	59.77%	153	44.61%

Table 9: Observed distribution of classes of code injection techniques implemented by malware in the sample sets from 2017, 2019 and 2020.

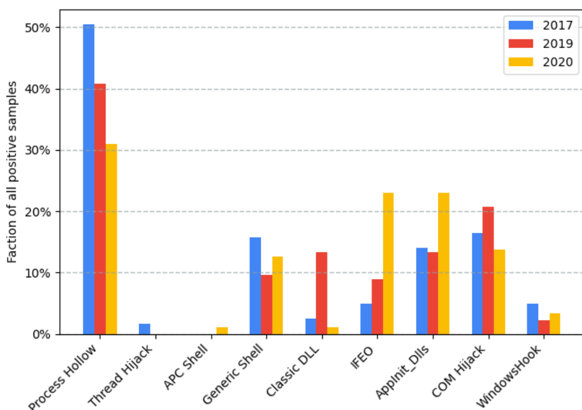

Figure 16: Observed distribution of code injection techniques by malware in the sample sets from 2017, 2019 and 2020. This graph only includes the techniques for which at least one sample was detected.

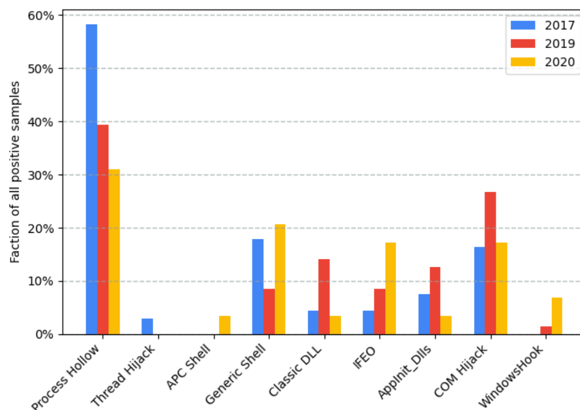

Figure 17: Observed distribution of code injection techniques by malware in the sample sets from 2017, 2019 and 2020, after normalizing by malware family.

ing the examination phase. One reason could be that the sample is waiting for a command from a remote server before it does anything nefarious. If this server does not exist any more or has been neutralized by authorities such as law enforcement, this means the sample will never activate itself, and never get to performing a code injection. Naturally, the probability that this is the case for a single sample increases the older the sample is. Therefore, statistics pulled from older sets (such as the one from 2017) might be an underrepresentation of reality.

A third reason for malware not activating itself, might be because malware sometimes is put inside an installer for another (benign) software package. The problem with this is that installation of software on Windows usually requires some form of interaction, such as the press of the "Next" button in an installation wizard. Since our examination environment does not support automatically performing these kinds of interactions, malware might not have gotten the chance to activate itself, because the installation never completed.

It could also be the case that the malware simply does not run properly on Windows 10. In 2017, Windows 7 was the most popular operating system in use [3]. It makes therefore sense for an attacker

to build malware that is compatible with this operating system specifically. However, changes between Windows 7 and 10 might have introduced incompatibilities with a sample and our examination environment. Therefore, similar to the case described before, some samples might not have run till completion, and as such, statistics calculated from older sample sets might be below the real numbers.

Finally, a malware sample might be activating itself and contain a procedure that performs an injection, but still decides against actually executing this injection. This is often the case for samples that use code injection as a form of privilege escalation. If a sample successfully injects malicious code into a process with higher privileges (e.g. administrator rights), then the malicious code will then also be executed under these higher privileges [49]. However, if the sample is already granted these rights, then the injection is in this use-case unnecessary and can be skipped. Some samples therefore will only perform the injection if they are below the desired level of privileges. Since in our examination environment we disabled UAC, every sample that we execute runs under these higher privileges from the moment the process starts. This might make these samples not perform an injection even though there is code for it present, and as such our system would incorrectly flag it as a negative.

7.2 Distribution of Techniques

A clear result that we can see in Figure 16 is that *Process Hollowing* is dominating convincingly across all three years. Since this technique is one of the more well-known methods, and the majority of malware developers tend to copy code from others [24], this is an expected result. However, this trend seems to go down in 2019 and 2020, while other techniques gain more popularity. This might be an indication that malware developers have started using less traditional methods for detection avoidance. The more well-known a technique is, the higher the probability that an anti-malware company will try to detect this type of behavior. Therefore, using lesser known methods increases the chances that malware will stay undetected for longer periods of time, making Process Hollowing not as much of a viable solution any more for malware developers as it used to be.

Despite the popularity of Process Hollowing, we can also see from Table 9 that the use of more passive techniques are a common choice as well. This is most likely because passive techniques often have a second purpose that is different from just getting code to run in the context of another process. For example, AppInit_DLLs Injection changes specific settings stored in the Windows Registry to let Windows load additional (malicious) libraries upon starting a new process. Since these settings are stored on the disk, they will survive a reboot of the system. This technique is therefore not just a method to perform code injection, but also a way to achieve persistence.

8 Limitations

During the development of the behavior nets and our detection system, we encountered a couple of limitations that our current implementation has.

The first main limitation is that while the theoretical model of Behavior nets itself is fairly generalized, the implementation might still be too specific and not universal enough to be able to recognize slight mutations of some forms of the techniques. One example of such a case is a model that aims to detect a technique falling under the configuration-based injection class. What makes it difficult to make these types of models as generic as possible, stems from the fact that configuration-based injections access very specific keys in the Windows Registry, and thus require the use of very specifc paths. While we can match on system calls such as `NtSetValueKey` with those paths as arguments, this does not encapsulate the underlying core characteristic sufficiently. Instead, we are matching on a change in the Registry at exactly this path, rather than the general concept of having a malicious file being loaded as a result of a malicious configuration. If another technique is found that uses a different registry key, then a new model will have to be created that matches on this new key.

Another limitation that touches on the previously mentioned issue, is that our implementation for the transition functions may be too specific as well. The theoretical model leaves the definition of the transition function fairly abstract; As long as it conforms

to a certain contract (it takes an event and token as input, and produces a new token as output), it is able to link events together and discard the noise in an event stream. However, in our implementation we always use a transition function that matches on specific API calls. This has the downside that if we want to match on events that can be implemented in multiple ways, we need multiple transition nodes in our behavior nets to be able to match on all of the options. This limitation can be seen in for example Figure 8, where place p_2 branches into two transitions matching either on the `CreateRemoteThread` or `NtCreateThreadEx` function. One potential improvement that could be made to our system to overcome this, is adding a preprocessing phase that lifts events in the event stream into classes of events before it is fed through the behavior nets. Alternatively, we could extend our DSL to allow for matching on multiple different types of events within a single transition block, and build up these equivalence classes directly in the behavior nets themselves. Both options would allow us to match on slightly higher abstraction of the type of events that we observe, and avoid the additional complexity in our nets all together.

9 Related Work

In this section, we relate our work with previously conducted research, and discuss where our system overcomes some of the shortcomings that this previous research has.

9.1 Existing Classifications for Code Injection

The idea of identifying common characteristics and placing code injection techniques in classes is a relatively new concept. Barabosch and Padilla were one of the first to make an attempt in identifying the key components of host based code injection techniques in 2014 [13]. In their work, they introduce two forms of injections; the *targeted* and *shotgun* approach, and two forms of code execution; *concurrent execution* and *thread manipulation*. These two concepts are similar to how we describe the *process-* and *thread model*. While these types of definitions are a very good starting point on their own, they do not provide much granularity. They put heavy focus on *what* the two forms of code injections entail, but not *how* to decide for any given thread whether it is a thread originating from the original application, or from the injected code. Furthermore, their work puts heavy focus on threads alone. It is true that running code always requires the context of a thread to run in, and therefore the invocation of injected code can always be classified as either the creation of a new thread, or the redirection of an existing thread. However, as mentioned in section 3, there exist a lot of different ways to perform the code execution in both classes, including techniques that do not communicate with the target process directly.

9.2 Existing Solutions for Automated Behavior Analysis

Similar to our solution, most fully automated systems that perform behavior analysis rely on dynamic analysis [2, 41, 15]. While these systems have been very successful and thorough with their examination, they usually stop at providing an examination report that is full of raw data, and do not always provide a good interpretation for it. The final verdict is often left to the analyst themselves.

Martignoni et al refer to this problem as the *semantic gap*. and have attempted to address this by constructing higher level abstractions from recorded behavior [44]. They do this by building up a set of what they call *behavior graphs*, which are quite similar to our behavior nets. Using these models, they were successful in detecting activity such as creating and executing files, downloading files, sending e-mails, and logging keystrokes.

Schneider proposed in 2000 the concept of *security policies*. Similar to our models, security policies are a form of automata that operate on a stream of execution steps (which we call events in our solution), and can be used to detect anomalies in the normal execution of a program. While this can be useful from a victim process' standpoint, it does assume that the original (correct) behavior of the victim process is known. Since we are monitoring an entire system with lots of complex closed source applications run-

ning in the background, this task becomes infeasible in our case. One could argue that a security policy could also be used to model the code injection techniques themselves, but since it would be expressed using a "normal" automaton, it would not be able to recover from arbitrary reordering of steps as described in section 4.1.1 without also enumerating all possible permutations.

Next to their formal models for code injection, Barabosch et al also proposed an automated method for deciding whether code was injected into another process [12]. In this work, they apply the honeypot paradigm by imitating legitimate processes such as a web browser, and deliberately allowing malware to inject in these decoy processes. While they show it can be quite effective, their system has limitations. For one, it heavily relies on the fact that the malware is identifying these decoy processes as potential targets. Furthermore, they also describe the limitation of not being able to control child processes that a sample might spin up themselves. This is similar to the limitations introduced by Cuckoo Sandbox [2], as it only provides a narrow window for a detector to monitor the system. Our solution, on the other hand, monitors the entire system, and thus does not have this restriction.

10 Conclusion

We have conducted a systematic study on code injection techniques, and proposed a taxonomy which groups these techniques into classes based on a set of common traits. We continued by introducing our extension to the Petri Net modelling language called *Behavior Nets*, which allows us to describe the techniques in terms of observable events and the dependency relations between them. We then presented a system that implements these nets to automatically determine whether an arbitrary sample uses code injection or not. We used this system to collect empirical evidence on the general prevalence of code injection, as well as the distribution of the used techniques in the malware scene of 2017, 2019 and 2020. Our experiments show that our system is capable of detecting various code injection techniques, and that at least 11.15% of all examined samples performed some form of code injection. Furthermore, the data suggests that Process Hollowing is the most commonly used technique. However, it also shows an indication of a shift in trend. More traditional techniques seem to be getting less used, while others become more prevalent. We have made our algorithms, as well as our test files and implementations of the studied code injection techniques, open source for the sake of open science.

11 Acknowledgements

We are very grateful to our supervisors Andrea Continella, as well as Marieke Huisman, who have provided us with valuable insights and the necessary feedback and proofreading.

References

[1] BPKT - ARM Compiler toolchain Assembler Reference. `https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/miscellaneous-instructions/bkpt`. Accessed: 2021-07-20.

[2] Cuckoo Sandbox - Automated Malware Analysis. `https://cuckoosandbox.org/`. Accessed: 2021-20-07.

[3] Desktop Operating System Market Share Worldwide. `https://gs.statcounter.com/os-market-share/desktop/worldwide/`. Accessed: 2021-07-20.

[4] Haskell Language. `https://www.haskell.org/`. Accessed: 2021-07-20.

[5] ld.so - Linux manual page. `https://www.man7.org/linux/man-pages/man8/ld.so.8.html`. Accessed: 2021-07-20.

[6] Malware Statistics & Trends Report. `https://www.av-test.org/en/statistics/malware/`. Accessed: 2021-09-09.

[7] The DOT Language. `https://www.graphviz.org/doc/info/lang.html`. Accessed: 2021-07-20.

[8] VirusTotal Malware Academic Dataset. https://www.virustotal.com/. Accessed: 2021-09-09.

[9] Y. A. Ahmed, M. A. Maarof, F. M. Hassan, and M. M. Abshir. Survey of Keylogger technologies. *International Journal of Computer Science and Telecommunications*, 5(2), 2014.

[10] C. Ammann. Hyperion: Implementation of a PE-Crypter. https://www.exploit-db.com/docs/english/18849-hyperion-implementation-of-a-pe-crypter.pdf, 2012. Accessed: 2021-01-28.

[11] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. *CS701 Construction of Compilers*, 19, 2005.

[12] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla. Bee Master: Detecting Host-Based Code Injection Attacks. pages 235–254, 2014.

[13] T. Barabosch and E. Gerhards-Padilla. Host-based code injection attacks: A popular technique used by malware. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 8–17, 2014.

[14] C. Barria, D. Cordero, C. Cubillos, and R. Osses. Obfuscation procedure based in dead code insertion into crypter. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*, pages 23–29, 2016.

[15] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[16] J.-M. Borello and L. Mé. Code Obfuscation Techniques for Metamorphic Viruses. *Journal in Computer Virology*, 4:211–220, 2008.

[17] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. SSYM'03, page 12, USA, 2003. USENIX Association.

[18] C. Cifuentes and K. J. Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[19] T. Cipresso and M. Stamp. *Software Reverse Engineering*, pages 659–696. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[20] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *2010 IEEE Symposium on Security and Privacy*, pages 61–76. IEEE, 2010.

[21] Continella, Andrea, and Carminati, Michele and Polino, Mario and Lanzi, Andrea and Zanero, Stefano and Maggi, Federico. Prometheus: Analyzing WebInject-based Information Stealers. *Journal of Computer Security*, 1, 2017.

[22] K. D. Cooper and L. Torczon. *Engineering a compiler*. Amsterdam: Elsevier/Morgan Kaufmann, 2012.

[23] Cyberbit. COM Hijacking – Windows Overlooked Security Vulnerability. https://www.cyberbit.com/blog/endpoint-security/com-hijacking-windows-overlooked-security-vulnerability/. Accessed: 2021-08-10.

[24] F. de la Cuadra. The geneology of malware. *Network Security*, 2007(4):17–20, 2007.

[25] Dejan Lukan. Using CreateRemoteThread for DLL Injection on Windows. https://resources.infosecinstitute.com/topic/using-createremotethread-for-dll-injection-on-windows/. Accessed: 2021-07-20.

[26] Dejan Lukan. Using CreateRemoteThread for DLL Injection on Windows. https://resources.infosecinstitute.com/topic/using-createremotethread-for-dll-injection-on-windows/. Accessed: 2021-08-10.

[27] Dejan Lukan. Using SetWindowsHookEx for DLL Injection on Windows. https://resources.infosecinstitute.com/topic/using-setwindowshookex-for-dll-injection-on-windows/. Accessed: 2021-08-10.

[28] S. Dong, M. Li, W. Diao, X. Liu, J. Liu, Z. Li, F. Xu, K. Chen, X. Wang, and K. Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International Conference on Security and Privacy in Communication Systems*, pages 172–192. Springer, 2018.

[29] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. volume 27, pages 1165–1178. IEEE, 2008.

[30] I. Dychka, I. Tereikovskyi, L. Tereikovska, V. Pogorelov, and S. Mussiraliyeva. Deobfuscation of computer virus malware code with value state dependence graph. In *International Conference on Computer Science, Engineering and Education Applications*, pages 370–379. Springer, 2018.

[31] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. 44(2), Mar. 2008.

[32] Elastic. Hunting In Memory. https://www.elastic.co/blog/hunting-memory. Accessed: 2021-08-10.

[33] F-Secure. Hunting for Application Shim Databases. https://blog.f-secure.com/hunting-for-application-shim-databases/. Accessed: 2021-08-10.

[34] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.

[35] H. Fang, Y. Wu, S. Wang, and Y. Huang. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security*, pages 168–181. Springer, 2011.

[36] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 9 2016. Volume 2 (2A, 2B, 2C 2D): Instruction Set Reference, A-Z.

[37] iRed.team. Import Adress Table (IAT) Hooking. https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking. Accessed: 2021-08-10.

[38] J. Keane. Sleeper Locker Ransomware Comes Alive, Infects Hundreds. https://www.digitaltrends.com/computing/sleeper-locker-ransomware-comes-alive-infects-hundreds/, 2015. Accessed: 2021-07-20.

[39] A. Küchler, A. Mantovani, Y. Han, L. Bilge, and D. Balzarotti. Does every second count? time-based evolution of malware behavior in sandboxes. In *Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society*, 2021.

[40] T. László and Á. Kiss. Obfuscating C++ programs via control flow flattening. volume 30, pages 3–19, 2009.

[41] 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 *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, page 386–395, New York, NY, USA, 2014. Association for Computing Machinery.

[42] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti. Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In *Network and Distributed System Security (NDSS) Symposium, NDSS*, volume 20, 2020.

[43] J. A. Marpaung, M. Sain, and H.-J. Lee. Survey on malware evasion techniques: State of the art

and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT)*, pages 744–749. IEEE, 2012.

[44] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. pages 78–97, 2008.

[45] Microsoft. Understanding Shims. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-7/dd837644(v=ws.10). Accessed: 2021-07-20.

[46] MITRE ATT&CK. Process Injection: Asynchronous Procedure Call. https://attack.mitre.org/techniques/T1055/004/. Accessed: 2021-08-10.

[47] MITRE ATT&CK. Event Triggered Execution: AppCert DLLs. https://attack.mitre.org/techniques/T1546/009/. Accessed: 2021-08-10.

[48] MITRE ATT&CK. Event Triggered Execution: AppInit DLLs. https://attack.mitre.org/techniques/T1546/010/. Accessed: 2021-08-10.

[49] MITRE ATT&CK. Privilege Escalation. https://attack.mitre.org/tactics/TA0029/. Accessed: 2021-09-09.

[50] MITRE ATT&CK. Process Injection: Dynamic-link Library Injection. https://attack.mitre.org/techniques/T1055/001/. Accessed: 2021-08-10.

[51] MITRE ATT&CK. Process Injection: Extra Window Memory Injection. https://attack.mitre.org/techniques/T1055/011/. Accessed: 2021-08-10.

[52] MITRE ATT&CK. Process Injection: Process Hollowing. https://attack.mitre.org/techniques/T1055/012/. Accessed: 2021-08-10.

[53] MITRE ATT&CK. Process Injection: Thread Execution Hijacking. https://attack.mitre.org/techniques/T1055/011/. Accessed: 2021-08-10.

[54] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[55] H. Orman. The Morris worm: a fifteen-year perspective. volume 1, pages 35–43, 2003.

[56] Pieter Arntz. An Introduction to Image File Execution Options . https://blog.malwarebytes.com/101/2015/12/an-introduction-to-image-file-execution-options/. Accessed: 2021-08-10.

[57] PreEmptive. What is obfuscation? https://www.preemptive.com/obfuscation. Accessed: 2021-07-20.

[58] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *2012 IEEE Symposium On Security And Privacy*, pages 65–79. IEEE, 2012.

[59] H. Saıdi, P. Porras, and V. Yegneswaran. Experiences in malware binary deobfuscation. *Virus Bulletin*, 2010.

[60] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? volume 49, pages 1–37. ACM New York, NY, USA, 2016.

[61] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero. Avclass: A tool for massive malware labeling. In F. Monrose, M. Dacier, G. Blanc, and J. Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing.

[62] Sevagas. PE Injection Explained. https://blog.sevagas.com/PE-injection-explained. Accessed: 2021-08-10.

[63] T. Shields. Anti-debugging–a developers view. *Veracode Inc., USA*, 2010.

[64] V. G. Tasiopoulos and S. K. Katsikas. Bypassing antivirus detection with encryption. In *Proceedings of the 18th Panhellenic Conference on Informatics*, pages 1–2, 2014.

[65] J. Wyke. The ZeroAccess Botnet Mining and Fraud for Massive Financial Gain. 2012.

[66] W. Yan, Z. Zhang, and N. Ansari. Revealing packed malware. volume 6, pages 65–69, 2008.

[67] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

A Behavior Net DSL Example

Listing 1: A Behavior Net expressed using our DSL, modelling the Process Hollowing technique. This net is equivalent to the net depicted in Figure 12.

```
behavior "Process Hollowing" {
  place [p0 p1 p2 p3 p4 p5 p6 p7]
  place p8 accepting

  transition t0 {
    NtCreateUserProcess(processHandle, threadHandle, _,_,_,_,_,_,_,_,_)
  }

  transition t1 {
    NtUnmapViewOfSection(processHandle, _)
  }

  transition t2 {
    NtAllocateVirtualMemory(processHandle, baseAddress, _, size, _, _)
  }

  transition t3 {
    NtWriteVirtualMemory(processHandle, address, _, _, _)
    where
      address in [baseAddress..(baseAddress+size)]
  }

  transition t4 {
    NtGetContextThread(threadHandle, _)
  }

  transition t5 {
    NtSetContextThread(threadHandle, _)
  }

  transition t6 {
    NtResumeThread(threadHandle, _)
  }

  t0 -> p0 -> t1 -> p1 -> t5
  t0 -> p2 -> t2 -> p3 -> t3 -> p4 -> t5
  t0 -> p5 -> t4 -> p6 -> t5
  t5 -> p7 -> t6 -> p8
}
```