Behavior Nets: Context-Aware Behavior Modeling for Code Injection-Based Windows Malware

JERRE STARINK, Semantics, Cybersecurity & Services, University Twente Faculty EEMCS, Enschede, Netherlands
MARIEKE HUISMAN, Formal Methods and Tools, University Twente Faculty EEMCS, Enschede, Netherlands
ANDREAS PETER, Departement of Computer Science, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany
ANDREA CONTINELLA, Semantics, Cybersecurity & Services, University Twente Faculty EEMCS, Enschede, Netherlands

Despite significant effort put into research and development of defense mechanisms, new malware is continuously developed rapidly, making it still one of the major threats on the Internet. For malware to be successful, it is in the developer's best interest to evade detection as long as possible. One method in achieving this is using Code Injection, where malicious code is injected into another benign process, making it do something it was not intended to do.

Automated detection and characterization of Code Injection is difficult. Many injection techniques depend solely on system calls that in isolation look benign and can easily be confused with other background system activity. There is therefore a need for models that can consider the context in which a single system event resides, such that relevant activity can be distinguished easily.

In previous work, we conducted the first systematic study on code injection to gain more insights into the different techniques available to malware developers on the Windows platform. This paper extends this work by introducing and formalizing Behavior Nets: A novel, reusable, context-aware modeling language that expresses malicious software behavior in observable events and their general interdependence. This allows for matching on system calls, even if those system calls are typically used in a benign context. We evaluate Behavior Nets and experimentally confirm that introducing event context into behavioral signatures yields better results in characterizing malicious behavior than the state of the art. We conclude with valuable insights on how future malware research based on dynamic analysis should be conducted.

CCS Concepts: • **Security and privacy → Malware and its mitigation;**

Additional Key Words and Phrases: Malware, software analysis, malicious behaviors, code injection

This work has been partially supported by the Mercedes project, Grant No. 639.023.710, funded by The Netherlands Organisation for Scientific Research (NWO).
Authors' Contact Information: Jerre Starink, Semantics, Cybersecurity & Services, University Twente Faculty EEMCS, Enschede, Overijssel, Netherlands; e-mail: j.a.l.starink@utwente.nl; Marieke Huisman, Formal Methods and Tools, University Twente Faculty EEMCS, Enschede, Overijssel, Netherlands; e-mail: m.huisman@utwente.nl; Andreas Peter, Departement of Computer Science, Carl von Ossietzky Universität Oldenburg, Oldenburg, Niedersachsen, Germany; e-mail: a.peter@utwente.nl; Andrea Continella, Semantics, Cybersecurity & Services, University Twente Faculty EEMCS, Enschede, Overijssel, Netherlands; e-mail: a.continella@utwente.nl.

ACM 2471-2566/2025/08-ART33
https://doi.org/10.1145/3729228

ACM Reference Format:
Jerre Starink, Marieke Huisman, Andreas Peter, and Andrea Continella. 2025. Behavior Nets: Context-Aware Behavior Modeling for Code Injection-Based Windows Malware. *ACM Trans. Priv. Sec.* 28, 3, Article 33 (August 2025), 29 pages. https://doi.org/10.1145/3729228

1 Introduction

Despite a significant effort put into research and development of defense mechanisms [11, 29], new malware is continuously developed at a rapid pace, making it still one of the major threats on the Internet [13].

For malware to be successful, it is in the author's best interest to make sure that their samples stay undetected for as long as possible [50]. One of the techniques that can be used to evade detection is *code injection*. Code injection is defined as the process in which an application copies pieces of its code into another running program. This running program is then tricked into executing the injected code, making it perform something it was not originally intended to [12, 16, 46]. By extension, if a malicious program copies its malicious code into a legitimate application, it is not the malware itself that exhibits the malicious behavior, but rather the application that was previously considered benign. As a consequence, scanning an executable file for suspicious code might not be sufficient, making the task of automating threat detection significantly more involved [50].

While code injection is often considered one of the main features of many malware families, the vast variety of code injection techniques is often overlooked by many sandboxes, which may lead to an incorrect assessment of the (malicious) capabilities of a sample: Existing solutions often rely on heuristics that look for specific byte patterns in the file [10, 22], the existence of suspicious memory pages in running processes [23, 26] or listen for calls to Windows APIs that are often associated with code injection [4, 5, 12, 15]. A main limitation of these heuristics is that many types of malicious behavior, including code injection, often cannot be reduced to a single API call or other observable system event. Instead, many tactics often comprise a sequence of carefully chosen API calls that in isolation look benign but when considered together become malicious. For example, the three Windows APIs `NtAllocateVirtualMemory`, `NtWriteVirtualMemory`, and `NtCreateThread` are commonly used in operations involving memory allocations, memory manipulations, and the creation of threads respectively in benign processes. However, when given specific arguments and called one after the other, they can form the basis of many code injection techniques. This makes detection and characterization of malicious behavior difficult, as it is not always clear whether a call to one of these APIs is part of a chain of events, or simply part of background noise.

In prior work [56], we conducted, to the best of our knowledge, the first systematic study on code injection to gain more insights into the different techniques available to malware developers on the Windows platform. We identified 17 different code injection techniques and categorized them in a taxonomy based on their common requirements and characteristics. This showed that many techniques operate fundamentally differently from each other, and indeed often require multiple (benign) system calls for them to manifest. Leveraging our taxonomy, we then proceeded by measuring the prevalence of these techniques in the general malware scene for the years 2017 and 2021 and found that there is an upward trend towards what we call *passive* techniques. These techniques almost exclusively make use of very benign-looking APIs in sequence and let the underlying operating system itself do most of the heavy lifting. This further stresses the need for a better characterization system that can deal with similar malicious behavior.

In our previous study, to detect the use of any of these code injection techniques, we prototyped a graph-based solution in our measurement framework inspired by dependency graphs [35] and

Petri Nets [45]. In this extension paper, we aim to generalize this solution by formalizing it into a novel, reusable modeling language which we call *Behavior Nets*. Behavior Nets describe malicious software behavior in terms of observable system events (such as API and system calls) with a particular focus on their interdependence. By introducing constraints on the arguments that each event is invoked with, a Behavior Net can be made aware of the *context* in which a single event resides. We then use this to identify and relate dependent events relevant to the malicious behavior and disregard other events originating from background (benign) system activity. We evaluate the effectiveness of this approach and experimentally confirm that Behavior Nets are more effective in reliably characterizing malicious behavior, in comparison with strategies often employed by other commonly adopted sandbox solutions. We conclude by providing insights on how future malware analysis research based on dynamic analysis should be conducted.

In short, we extend our prior work [56] with original concepts, discussions, and new results, and we make the following contributions with respect to our previous paper:

- **Behavior Nets:** We design and formally specify a novel, context-aware modeling language, Behavior Nets, to characterize code injection techniques in terms of the required observable system events and their interdependence. We also provide a reference implementation including a **domain-specific language (DSL)** to easily and concisely define other types of behavior.
- **Comparison with existing signature models:** We provide an extensive evaluation of the effectiveness of Behavior Nets by comparing it to existing behavior fingerprinting techniques used in commonly available sandbox solutions.
- **Insights for future malware behavioral research:** We conclude by providing important insights on how research based on malware behavioral analysis should be reliably conducted in the future.

In the spirit of open science, we publish all our code and findings at https://github.com/utwente-scs/behavior-net.

2 Background

We begin by revisiting the concept of code injection, and reintroducing the terms that were also introduced in [56].

2.1 Code Injection Fundamentals

Code injection can be defined as the act of copying and executing code in the context of another process. An *injector* typically starts by selecting one or more *victim* processes to inject into. Victim processes can be any process running on the system, or a new process that the injector itself starts up. The injector proceeds by finding either existing writeable memory pages already present in the victim process or may allocate new ones, to then copy new code into — often referred to as the *payload*. Finally, the injector ensures the memory pages the payload is copied into are marked executable and then tricks the victim process into executing it. Ultimately, the goal of code injection is to alter the behavior of the victim process, making it do something it is not intended to do.

Injecting code into another process is an effective way to hide the true (malicious) intentions of a program. Detection mechanisms that solely focus on analyzing the sample itself might not pick up on the behavior offloaded to the victim process. Especially victim processes from a known vendor are an attractive option for an injector process, as these programs are often blindly trusted by anti-malware [16, 46]. For these reasons, several variants of code injection have been adopted by modern malware as a detection evasion technique, and are often recognized as a main feature a malware family is often characterized with by security vendors [25, 49, 60].

Unfortunately, fully abolishing the use of code injection is not a practical solution for mitigating the threat code injection brings. This is because several types of legitimate software use code injection in benign contexts as well. For example, many *debuggers* rely on injecting small chunks of code into the target process to stop its execution (typically using instructions that trigger breakpoint trap exceptions [1, 27]) and then read its internal state. Additionally, many operating systems feature *shim infrastructures* to make up for incompatible version updates. These are implemented by hooking into API functions provided by the underlying operating system and redirecting them to injected *shim* code that simulates the original behavior of the API before a breaking change [36]. Finally, as was demonstrated in [56], some accessibility programs such as the virtual display keyboard on the Microsoft Windows operating system may use code injection to simulate keystrokes and other types of inputs and inject them into the event loop of other applications. Prohibiting code injection would thus mean giving up on these use cases and frameworks.

2.2 Code Injection Techniques

In our previous work [56], we queried various sources that are well-known in the security community to obtain a representative set of code injection techniques. These include the MITRE framework, as well as technical malware briefings provided by six well-known security companies, including The Infosec Institute, Elastic Security, MalwareBytes, F-Secure, Symantec, and Kaspersky. We also included various blog posts of individual security researchers with example implementations and variations of the techniques. Since malware authors typically aim to maximize their attack surface, we select only the techniques that work on Windows 10 (as it is the most market-dominant OS at the time of conducting this research [57]), and do not have a dependency on extra (third-party) software that needs to be installed separately. With this process, we selected the following 17 techniques:

Shellcode Injection. This technique is the most fundamental form of code injection and serves as a base for many other techniques. First, the victim process is opened using a system call to `NtOpenProcess` and memory is allocated within this process using `NtAllocateVirtualMemory` with the `PAGE_EXECUTE_READWRITE` protection bit set. Then, shellcode is transmitted into this allocated memory using the `NtWriteVirtualMemory` function. Finally, a thread with the address of the injected shellcode as its entry point is created within the victim process, usually through an API such as `CreateRemoteThread` or using the underlying system call `NtCreateThreadEx` directly [23].

PE Injection. PE injection extends Shellcode Injection by including additional logic to support injecting entire **Portable Executable (PE)** files, the standard file format used on Windows to store binary compiled executable files. This additional logic prepares a payload that looks exactly like a PE as if it were mapped into memory by Windows itself, by manually aligning each section in the file to the right virtual addresses, resolving function addresses used by the PE, and applying any base relocations present in the headers. This allows for easier development of larger, more complex payloads written in higher-level languages as opposed to small (handcrafted) assembly code. As these extra steps in the payload preparation can be implemented fully without the need for adding additional system calls, the system call profile of this technique is identical to Shellcode Injection [55].

Classic DLL Injection. Direct calls to `NtAllocateVirtualMemory` with the `PAGE_EXECUTE_READWRITE` bit set are often considered suspicious by state-of-the-art [23, 26]. Classic DLL Injection avoids calls like these by first writing the payload into a **Dynamic-Link Library (DLL)** file on the disk instead. The injector then allocates some *non-executable* memory (i.e., `NtAllocateVirtualMemory` but with the `PAGE_READWRITE` bit set) to write the file path of the

newly created DLL into (NtWriteVirtualMemory). Then it leverages NtCreateThreadEx to create a new thread starting at LoadLibrary — a user-mode function provided by Windows itself responsible for loading DLL files dynamically — with its first argument set to the address of the injected file path. As a result, the victim process is tricked into calling LoadLibrary with the path to the payload DLL, loading and executing it as if it was a normal dependency. This approach avoids the allocation of suspicious executable memory pages, at the cost of requiring two extra system calls NtCreateFile and NtWriteFile to write the DLL to the disk [20, 42].

Reflective DLL Injection and Memory Module Injection. These two techniques are variations of Classic DLL Injection that add similar logic found in PE Injection to reimplement the functionality of LoadLibrary. This way, they avoid the call to the original function (which could be monitored) and can also keep the payload DLL in memory (avoiding the need for extra calls to NtCreateFile and NtWriteFile). Both injectors use Shellcode Injection to inject a payload into the victim process, giving them identical system-call profiles. The difference between the two techniques is that Reflective DLL Injection implements this manual mapping logic on the side of the victim process (i.e., the payload is mapping itself), while Memory Module Injection performs most of the manual mapping on the injector's side instead. Memory Module Injection also ensures that the mapped sections have the appropriate protection bits set (as opposed to only PAGE_EXECUTE_READWRITE), which contributes to the stealthiness of the technique [23].

APC Shell and DLL Injection. These two techniques are variations of Shellcode and Classic DLL Injection respectively, that avoid the creation of new threads using NtCreateThreadEx by abusing the **Asynchronous Procedure Call (APC)** queue of an *existing* thread instead. APCs are function calls that are scheduled to be invoked by a thread when the thread is in a *waiting state* (e.g., waiting for an event or user input). By replacing the NtCreateThreadEx call with a call to NtOpenThread and NtQueueApcThread, the injector can open an existing thread and queue shellcode or a LoadLibrary call as an APC, which makes Windows automatically load and trigger the execution of the payload whenever the thread is in such a state [38].

Process Hollowing and Thread Hijacking. These are one of the most commonly used methods for performing code injection and are sometimes also referred to as RunPE. The injector either creates a new suspended process (NtCreateUserProcess with the THREAD_CREATE_FLAGS_CREATE_SUSPENDED bit set) or suspends an existing one (NtOpenProcess followed by NtSuspendProcess or NtSuspendThread) respectively, and unmaps (hollows out) all its sections from memory (NtUnmapViewOfSection). Then, a new PE image is manually mapped into the victim process, similar to how it is done in PE Injection, and the main thread's program counter register is updated using NtSetContextThread to redirect the execution to the entry point of the injected PE. Finally, the process is resumed afterward using NtResumeThread [37, 44].

IAT Hooking. During the loading procedure of a PE file, Windows resolves the addresses of all functions that the PE depends on and puts them in its **Import Address Table (IAT)**. The IAT Hooking technique replaces one of these addresses with one that points to the injected shellcode, typically invoking NtWriteVirtualMemory. This way, when the victim process calls the original function using its IAT, the payload will be triggered instead, without using thread creation or redirection APIs, giving this technique a very low profile that is hard to detect on just a system call level [28].

CTray VTable Hooking. This technique is similar to IAT Hooking but specifically targets explorer.exe, the default file browser on Windows. Internally, the browser defines a class CTray which implements the taskbar's notification tray. By replacing the address of its WndProc function, which is responsible for processing every message that the tray receives (e.g., paint events), the technique activates injected shellcode the moment the tray processes such a message. In contrast

to IAT Hooking however, this technique does require an extra system call to `NtUserFindWindowEx` to find the window this `CTray` class is assigned to. Additionally, a call to `NtUserSetWindowLong` or `NtUserSetWindowLongPtr` is required to reassign the address of `WndProc`. [43].

Shim Injection. Shim infrastructures are small programs attached to legacy software, that attempt to simulate the original behavior of an API after a breaking change was introduced by a Windows update. By extension, an injector can register itself as a shim infrastructure to load and run arbitrary code within the context of software that requires these legacy features. It does so by adding itself in the Windows Registry at a specific path using `NtSetValueKey` [24].

Image File Execution Options (IFEO). Image File Execution Options (IFEO) are settings stored in the Windows Registry that dictate how a specific application identified by its name should be started by Windows. One of the parameters it defines is the path to a debugger program that the application's memory should be replaced with when it is being loaded. Similar to Shim Injection, IFEO Injection only requires adding itself as an entry in the Windows Registry using `NtSetValueKey` to register itself as such a debugger and thus redirect execution to the payload [48].

AppInit_Dlls and AppCertDlls Injection. Similar to IFEO, AppInit_Dlls and AppCertDlls are two Windows Registry keys that store the paths to extra DLL files that should be loaded whenever an application starts. The key difference is that these DLLs are loaded by *any* process that is started after the Registry change was made, as opposed to specific processes [39, 40].

COM Hijacking. The **Common Object Model (COM)** is a Windows framework that allows for software components to be used across multiple programming languages. Components are stored in the Windows Registry as file paths to the DLLs that implement them and are loaded and instantiated on-demand. COM Hijacking replaces one of these DLL file paths with a path of its own, tricking the victim process into loading the payload DLL instead of the original component [41].

Windows Hook Injection. The Windows API exposes functions to subscribe to various global system events such as mouse clicks and key presses. More specifically, by calling `NtUserSetWindowsHookEx`, a thread can be instructed to invoke a callback defined in a specific DLL when such an event occurs. Typically, threads are chosen from the current process. However, `NtUserSetWindowsHookEx` takes in as argument a thread ID that allows for selecting *any* thread running on the system. Windows Hook Injection abuses this by registering a callback for one of the victim process threads to a function defined in a DLL of its own, letting it load and execute a payload DLL [21].

2.3 Common Characteristics

From the studied techniques, we extracted common features that helped us characterize the techniques more precisely in [56], which we will revisit below.

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

Transmitter. The transmitter is the process that is responsible for copying the code into the victim process. For many techniques, this is done by the injector process itself, usually through a call to `NtWriteVirtualMemory`. However, some techniques trick the victim process into loading the code instead, e.g., by letting it read a malicious file.

Catalyst. The catalyst is the process responsible for triggering the execution of the injected code. Similar to the *Transmitter*, this is often done on the injector's side, e.g., by creating a thread within the victim process. Alternatively, the victim may also be tricked into calling the injected code itself.

File Dependency. A good amount of techniques require a copy of the injected code on the disk, usually in the form of a **Dynamic Link Library (DLL)**. This means that such a file needs to be stored before execution can take place.

Shellcode Dependency. Some techniques require a small chunk of code to be injected directly into the victim process to execute the final payload.

Process and Threading Model. These two traits describe how malware selects and interacts with the victim process and its threads. Some techniques interact with already running processes or threads, while others spawn new ones. Alternatively, some techniques rely on the operating system itself and do not directly interact with any process or thread at all.

Memory Manipulation Model. This describes the dependency on directly allocating or manipulating the memory of the victim process. It is often accompanied by opening a process first and is present in most classic techniques.

Configuration Model. Some injection techniques depend on changing specific settings of the victim process or underlying OS. They may alter the Windows Registry, or install malicious plugins in a user application such as a web browser. Often, they also rely on the existence of a file on the disk.

2.4 Taxonomy

Using the identified traits, we define a taxonomy for code injection (Table 1) and discuss our classes below.

Active and Passive Injections. The most distinguishing feature that we observe deals with the level of interaction that is required by a code injection technique. Many techniques actively communicate with the victim process by creating or opening processes and threads and directly interacting with their memory. Since these kinds of interactions often translate to distinct sets of API calls, they can be observed by monitoring software more easily, which contributes to the stealthiness (or lack thereof) of the technique. Therefore, let us introduce the concept of *active* code injection techniques:

Definition 1 (Active Techniques). A code injection technique is **active** if it directly interacts with the victim process or one of its threads, or actively changes the victim process' memory.

Many existing techniques are active. 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 a system call such as `NtWriteVirtualMemory` [23]. In contrast, 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 OS load and execute the code instead [24]. Thus, *Shim Injection* is considered a *passive* technique.

Intrusiveness and Destructiveness. We can further subdivide 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 in a way that parts of the application or the entire process stop working properly. Therefore, let us introduce the notion of intrusive and destructive injection techniques:

Definition 2 (Intrusiveness). An active code injection technique is **intrusive** if it directly changes (parts of) the victim process' existing memory or threads.

Definition 3 (Destructiveness). A technique is **destructive** if it is intrusive and (parts of) the application stop(s) working due to the intrusive intervention.

Table 1. Taxonomy of Code Injection Techniques and their Characteristics

		Technique	Moment of Execution	Transmitter	Catalyst	File Dependency	Shellcode Dependency	Process Model	Threading Model	Memory Manipulation Model	Configuration Model	Functional on Windows 10
Active	Intrusive	Destructive	Process Hollowing [44]	P	I	I		✓	N	E	N	✓
			Thread Hijacking [37]	A	I	I		✓	E	E	N	✓
			IAT Hooking [28]	A	I	V		✓	E		E	✓
			CTray Hooking [43]	A	I	V		✓	E		E	✓
			APC Shell Injection [38]	A	I	V		✓	E	E	N	✓
			APC DLL Injection [38]	A	I	V	✓		E	E	N	✓
	Non-Intrusive		Shellcode Injection [23]	A	I	I		✓	E	N	N	✓
			PE Injection [55]	A	I	I		✓	E	N	N	✓
			Reflective DLL Injection [23]	A	I	I		✓	E	N	N	✓
			Memory Module Injection [23]	A	I	I		✓	E	N	E	✓
			Classic DLL Injection [20, 42]	A	I	I	✓		E	N	N	✓
Passive	Configuration		Shim Injection [24]	P	V	V	✓				✓	✓
			Image File Execution Options [48]	L	V	V	✓				✓	✓
			AppInit_DLLs Injection [40]	L	V	V	✓				✓	✓
			AppCertDLLs Injection [39]	L	V	V	✓				✓	✓
			COM Hijacking [41]	L	V	V	✓				✓	✓
			Windows Hook Injection [21]	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, thread or memory page creation, **E**: Existing process, thread or memory page manipulation.

An example of a destructive technique is *Process Hollowing*, which creates a new victim process in a suspended state and replaces the original memory content with new code [44]. As a result, upon resuming, the victim process does not perform its original activity anymore. This is in contrast with *Classic DLL injection*, which forces the victim to load an additional library from the disk without interrupting any threads or modifying their code [20]. Thus, *Classic DLL Injection* falls under the *non-intrusive* category.

Configuration-based Injections. A more fine-grained subdivision can be made in our class of passive code injection techniques. This subdivision groups together techniques that require specific changes in the Registry, and is a direct result of the *Configuration Model* trait. An example of such a technique is *AppInit_DLLs Injection*, which registers a library file into the Registry. On the other hand, the *Windows Hook* injection technique directly interfaces with system events and does not require a persistent configuration stored on the disk.

Summary and Implications. Our systematization shows that different code injection techniques take very different approaches to transmitting and executing code. As such, each technique has its own set of characteristics that a detection mechanism should take into account. Popular open-source sandboxes such as Cuckoo [5] and CAPE [4] implement detection mechanisms using API call tracing for most active techniques. They also include some more generic heuristics

Table 2. Two Recorded API Call Traces

(a) APC Shell Injection sample		(b) Unrelated sample	
Time	**Observed API call**	**Time**	**Observed API call**
t_i	`NtOpenProcess(0xA0, ...)`	t_j	`NtOpenProcess(0xD8, ...)`
t_{i+1}	`NtAllocateVirtualMemory(0xA0, ...)`	t_{j+1}	`NtAllocateVirtualMemory(0x10, ...)`
t_{i+2}	`NtWriteVirtualMemory(0xA0, ...)`	t_{j+2}	`NtWriteVirtualMemory(0x28, ...)`

One sample implements APC Shell Injection and one sample uses similar but unrelated function calls.

for detecting transmissions from one process to another by looking for API calls commonly associated with code injection (e.g., `NtWriteVirtualMemory`). However, the existence of passive techniques indicates that monitoring these common API calls might be insufficient. Most passive techniques are either not included in the signature database, or are not classified as a method of injection. Besides, since passive techniques leverage features of the underlying OS to perform their transmission and catalyst, the line between benign and injected memory pages becomes significantly more blurred—both types of pages come from the same *origin* and are allocated in the same way as normal pages. These important realizations indicate that more injections might be adopted in the malware scene than was previously thought.

3 Methodology

We now present our methodology to characterize software behavior in malware samples, with the starting point of code injection techniques. Since malware developers often obfuscate or pack their samples, static analysis is not a feasible solution. We thus assume a system that treats the sample as a black box and can dynamically record all events and side effects observed in the system during the execution of the sample. We define a single event as a 2-tuple consisting of an identifier of the invoked event and a set of arguments the event was invoked with. A typical event is a call to a system function or service such as `NtAllocateVirtualMemory` or `NtWriteVirtualMemory`, which are the main functions on the Windows platform used to allocate and write memory into processes, respectively. We refer from now on to the recorded stream as the *event stream*.

It is important that the event stream is a recording of an entire system as opposed to a single process. This is because code injection is inherently a procedure that involves at least two processes (an injector and a victim process). Furthermore, there have been multiple cases of malware where the workload is split up into a collection of smaller tasks which either were distributed over various processes (including code injection itself) or were invoked one after the other as part of a "kill-chain" where each process fulfilled a single task [17, 34, 47]. As such, isolating behavior on a single process is therefore insufficient in fully capturing the workloads of modern malware. We therefore assume a trace that includes events originating from any process running on the system.

3.1 Challenges in Characterizing Malware Behavior

Finding evidence of malware behavior in a system-wide event stream recording requires overcoming two main challenges. The first challenge pertains to the stream containing a lot of "background noise". Events 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 need to be discarded. This is in particular the case for commonly used APIs such as `NtAllocateVirtualMemory`, which brings the problem of determining which calls to `NtAllocateVirtualMemory` are actually relevant to the behavior that we try to recognize, and which are part of standard behavior exhibited by the system itself (see Tables 2(a) and 2(b) for example traces).

The second challenge relates to code injection techniques (as well as many other types of software behavior) typically requiring multiple Windows API calls in sequence. For example, in many code injection techniques, a call to `NtAllocateVirtualMemory` is often followed by a call to `NtWriteVirtualMemory` to actually write the code into the previously allocated memory of the victim process. Additionally, the same technique may also require creating a file on the disk using, e.g., the system calls `NtCreateFile` and `NtWriteFile`, respectively. As the two API sequences are completely independent, a malware developer has lots of freedom in the exact order they could call them to get to their desired end result (e.g., one after the other, or some interleaved version of the two API sequences). This (intentional) reordering of independent steps in their implementation, combined with the general nature of observing concurrent systems, forces us to assume some non-determinism in the order in which certain APIs are called and thus appear in our event stream. This also means we cannot rely on a single sequence of events to look for in our event stream.

3.2 Key Intuition

Our approach to solving both challenges described in Section 3.1 relies on two key insights. Firstly, to address the background noise, we use the insight that related system calls will have similar if not identical arguments. This intuitively makes sense. A call to `NtAllocateVirtualMemory` with a certain process handle can only succeed if that same handle was opened before by a preceding call to `NtOpenProcess` or similar. Thus, an event stream containing these two calls will therefore also contain an occurrence of the same handle argument in both events and similar but unrelated events will use different arguments instead (as can be seen in Tables 2(a) and 2(b)).

Secondly, to solve the problem of non-determinism, we use the insight that recognizing behavior in a single event stream, where the exact order of independent operations does not matter but the general dependency does, is the same as recognizing behavior in a concurrent system where multiple independent processes run at the same time. Consider three threads T_A, T_B, and T_C, where T_A and T_B run concurrently and T_C waits for T_A and T_B to finish before it continues its execution. If we record the activity of these three running threads, we end up with an event stream that starts with an arbitrary interleaving of the events produced by T_A and T_B, and ends with the events of T_C in its entirety. Now consider another thread T_D, which performs the exact same operations of threads T_A, T_B, and T_C in this exact same order. What emerges is a resulting event stream that is indistinguishable from the stream we constructed earlier from the individual threads. This shows that modeling concurrent behavior is equivalent to modeling a single-threaded system where independent operations might be reordered in a non-deterministic manner.

We use both of these insights as a foundation for the design of our detection models.

3.3 Petri Nets

One method to model concurrent behavior is by using Petri Nets. Let us first recall the definition of a net:

Definition 4 (Net). A net is a bipartite graph defined by the tuple $N = (P, T, E)$, where P and T are disjoint finite sets of nodes, representing places and transitions respectively, and $E \subset (P \times T) \cup (T \times P)$ denotes the set of edges between these nodes.

Petri Nets are nets where places may contain several marks called *tokens* [45]. More formally:

Definition 5 (Petri Net). A Petri Net is a tuple $PN = (N, M)$, where $N = (P, T, F)$ is a net, $M : P \rightarrow \mathbb{N}$ a mapping that assigns a number of tokens to every place.

Figure 1 depicts an example Petri Net with four places and four transitions. In the initial state of this net (Figure 1(a)), we can see two tokens added to the places before the transitions t_2 and t_4

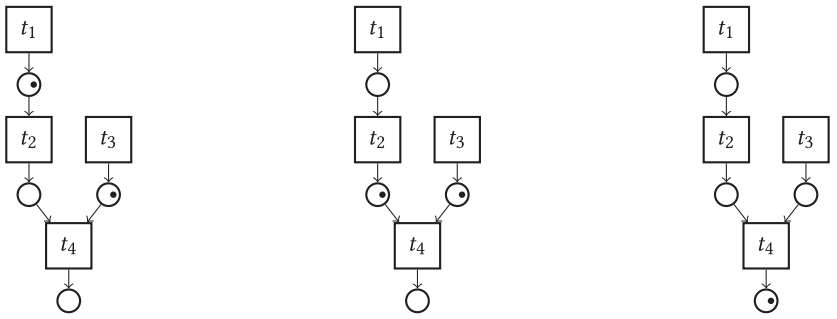


Fig. 1. The evolution of a Petri Net with four places and four transitions. By convention, places are represented by circles and transitions are depicted using rectangles. Two tokens, indicated by the black dots in place nodes, are flowing through the net as transitions t_2 and t_4 are being fired.

respectively (marked by two dots in the figure). The general idea of a Petri Net is that these tokens flow through the net as transitions are *fired* repeatedly. The firing of a transition consumes one token from all its input places and produces a new token in all of its output places. This can only happen when this transition is *enabled*, that is, when for each of its input places there are enough tokens present. In the initial state of the example net, t_2 is the only transition that is currently enabled. Indeed, only this transition has a token in all of its input places. After firing t_2, the token passes through t_2 and then disables t_2 as there are no more tokens present in its input places (Figure 1(b)). The power of Petri Nets lies in the fact that two edges in a net can converge into a single transition node (such as the ones at the edges towards t_4). Only after firing t_2, transition t_4 has enough tokens in all of its input places that can thus be fired (Figure 1(c)). Thus, this mechanism can model and evaluate a *thread barrier* or the *joining* of multiple threads, where a third workload waits for two other workloads to complete before continuing its execution.

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

3.4 Behavior Nets

We now extend the concept of Petri Nets and introduce a novel modeling language called Behavior Nets. Behavior Nets are very similar to Petri Nets, but are designed specifically for recognizing behavior patterns that can be precisely specified with event context in mind. In a Behavior Net, the transition nodes are labeled with an *event pattern* that is expected in the event stream. An event pattern consists of the expected event identifier and a set of constraints that need to be met before the transition is considered enabled and thus can be traversed by a token. These constraints can describe general bounds on arguments, but can also depend on values that were previously observed in other, dependent events. Places and transitions are connected by edges in such a way that they encode their general interdependence between the expected event patterns. Transitions are then fired for each event in the input event stream where applicable.

Figure 2 depicts an example Behavior Net describing the general behavior of the APC Shell Code Injection technique. In this net, we can see a chain of transition nodes labeled with the system events `NtOpenProcess`, `NtAllocateVirtualMemory` and `NtWriteVirtualMemory`, indicating that this order of events is to be expected in the event stream. Note that this does not mean this exact sub-sequence of events should appear in the event stream. Rather, edges encode general dependence between the events, and the chain could be interrupted by other independent system calls.

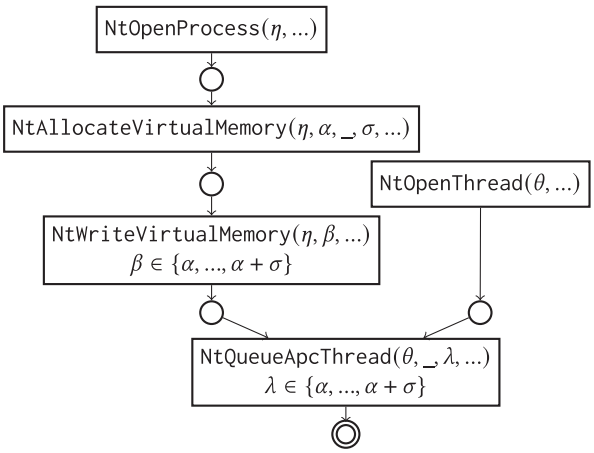


Fig. 2. A Behavior Net modeling *APC Shell Injection*. For brevity, we use underscores ('_') and ellipses ('...') to discard irrelevant parameters. The accepting state is indicated by a double outline.

This is further exemplified by the node labeled with the `NtOpenThread` system call pattern. As this node is not part of the same event chain on the left, it indicates an occurrence of `NtOpenThread` can happen before, during, or after the execution of the left event chain, and thus forms a secondary, independent chain of events. The two chains join together in a single transition node labeled with `NtQueueApcThread`, indicating that both independent chains must have been observed in their entirety, before the net can consider `NtQueueApcThread` events. This mechanism effectively solves the problem of dealing with non-determinism in the input event stream, as independent steps can be encoded without enumerating all possible orderings.

A second key addition to Petri Nets is that the event patterns in a Behavior Net are implemented using *transition functions* that operate on a set of *symbolic variables*. These variables are not part of the original program itself but are meant to capture parameters or a result of an event and exist within the net alone. Notice in Figure 2 the transitions for `NtOpenProcess`, `NtAllocateVirtualMemory` and `NtWriteVirtualMemory` have their first argument set to a symbolic variable η, indicating the observed first argument for all three system calls must be equal. We also use extra constraints on the `NtWriteVirtualMemory` transition to restrict the value of β to the interval $\{\alpha, \ldots, \alpha + \sigma\}$. This indicates that β should be a memory address that falls within memory that was previously allocated in the victim process by a call to `NtAllocateVirtualMemory`. Finally, the transition node matching on `NtQueueApcThread` also illustrates how the result of two independent API calls can be combined to express the catalyst of this technique, without assuming a specific order in which its dependent APIs were invoked. Here, θ represents a thread handle obtained from a prior call to `NtOpenThread`, and λ is an entry point address that is constrained to be within the allocated memory range. The use of symbolic variables makes the Behavior Net aware of the context an event resides in, and effectively solves the problem of distinguishing between relevant and background events.

Figure 3 depicts a more elaborate example of a Behavior Net modeling the Process Hollowing technique. This net showcases how nodes can also branch out into multiple parallel execution paths (similar to a fork-construction in asynchronous programming). As with normal Petri Nets, when the `NtCreateUserProcess` transition is fired, its three output places will all be populated with a copy of the resulting output token, each instantiating a new independent potential system call chain to be observed. We can also see that, similar to APC Shell Injection, these independent

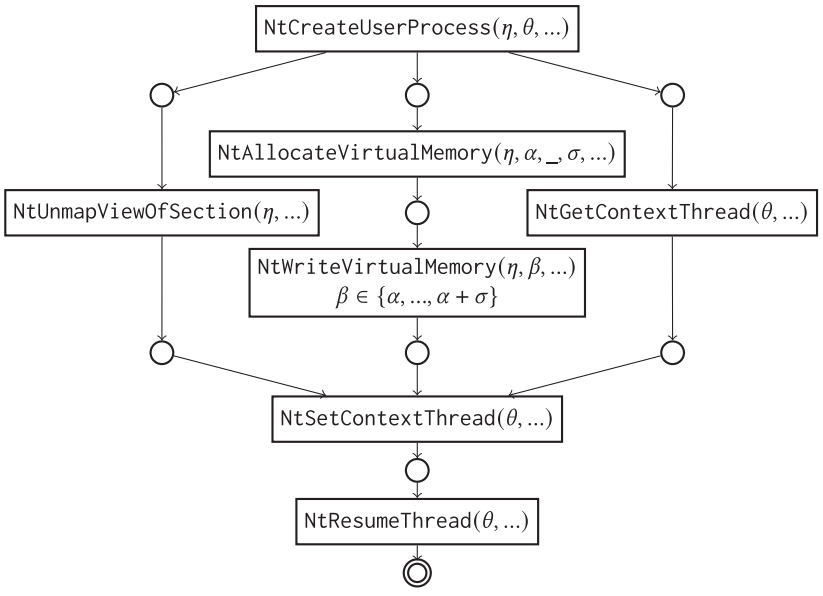


Fig. 3. A behavior net modeling the *Process Hollowing* technique. The `NtCreateUserProcess` transition branches into three different independent paths that need to be observed after the first system call is observed. The three branches also converge into the same node, indicating that all three paths must complete before `NtSetContextThread` is observed.

call chains all converge back into a single node, this time matching on `NtSetContextThread`. This indicates that the `NtSetContextThread` call is *dependent* on all three independent chains and thus should only be considered by the model if all of the chains were observed in their entirety.

Finally, similar to other types of automaton, Behavior Nets include accepting places, denoted in the figure with a double outline. When a token manages to move into an accepting place, the behavior is considered *recognized*.

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. We then define a Behavior Net as follows:

Definition 6 (Behavior Net). A Behavior Net is a tuple $B = (N, A, M, \delta)$, where

- N is a net,
- $A \subseteq P$ is the set of all accepting places,
- $M : P \to \mathcal{P}(\mathcal{T})$ is a marking; a mapping that assigns a set of tokens to every place in the net,
- $\delta : T \to (\Sigma \times \mathcal{T} \to \mathcal{T})$ is a mapping that assigns transition functions to every transition in the net.

In the remainder of this section, we will detail the exact execution semantics of Behavior Nets.

3.5 Event Patterns and Transition Functions

We now detail the exact role that tokens fulfill in a Behavior Net, and how they are used in transition functions to communicate event context and evaluate event patterns.

In a Behavior Net, a token τ is associated with a mapping between symbolic variables and their concrete observed values and thus can be seen as one possible instantiation or concretization of

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:          result ← {δ_t(e, ∅)}
5:      else
6:          result ← ∅
7:          Q ← {M(p)|p ∈ input places of t}
8:          for all (τ_1, ..., τ_n) ∈ Combinations(Q) do
9:              τ_combined ← TokenCombine(τ_1, ..., τ_n)
10:             τ_new ← δ_t(e, τ_combined)
11:             if τ_new ≠ ⊥ then
12:                 result ← r ∪ {τ_new}
13:     return result
```

all symbolic variables in the net. We use the notation $\tau = \{\alpha := x, \beta := y\}$ to denote that τ assigns the concrete values x and y to symbolic variables α and β, respectively. As transitions pull in their incoming tokens, they produce new tokens with updated mappings according to the event pattern they are annotated with. Transitions are then enabled if and only if there are enough tokens in its input places, and these input tokens are consistent with the constraints described in the event pattern.

In a Behavior Net, tokens define a TokenCombine operation. When two tokens τ_1 and τ_2 are combined, a new token is produced that stores the values of both original tokens. If τ_1 assigns a value to symbolic variable α that is different from the value in τ_2, we speak of τ_1 and τ_2 as tokens that are in *conflict*. Combining any conflicting tokens results in $\perp$, the invalid token. Combining any other token with $\perp$ also results in $\perp$.

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

Algorithm 1 describes the process of determining the new tokens when t is fired. Every combination of input tokens is first combined into a single token and then fed into 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 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 a single token is produced instead.

A token holding an instantiation of symbolic variables allows δ_t to decide whether a certain observation is part of a chain of events that we are interested in. Taking the first example described in Section 3.1, suppose δ_t matches on Windows API function calls to `NtWriteVirtualMemory`. Without also using an input token in our matching criteria, a call to this function used by a code injection would be indistinguishable from the ones introduced by background processes (see Tables 2(a) and 2(b) for example traces). However, by also considering the arguments that were used to call the function and trying to match them with the observed values stored in the incoming tokens, we can verify that the first argument (the process handle) matches an argument that was observed in prior calls to `NtAllocateVirtualMemory` or `NtOpenProcess`. By letting transition functions assign new values to symbolic variables in a token, they can then communicate this contextual information to other transitions in the net. This way, a Behavior Net can define constraints on event dependencies, decide which events are related to each other, and which can be filtered out.

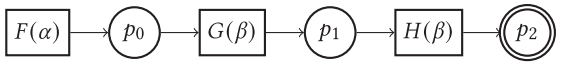


Fig. 4. A Behavior Net with three transitions matching on different events f, g, and h. The last two transitions share a symbolic variable β, indicating the arguments for both g and h need to be the same value.

Table 3. The Evolution of the Marking of the Behavior Net in Figure 4 with Event Stream $(F(x), G(y), G(z), H(z))$

(a) The evolution of the marking with token consumption. The model consumes the token $\{\alpha := x\}$ at t_{i+1}, resulting in the greedy assignment of $\beta := y$, causing the model to get stuck.

		Markings		
Time	**Event**	$M(p_0)$	$M(p_1)$	$M(p_2)$
t_i	$F(x)$	$\{\alpha := x\}$		
t_{i+1}	$G(y)$		$\{\alpha := x, \beta := y\}$	
t_{i+2}	$G(z)$		$\{\alpha := x, \beta := y\}$	
t_{i+3}	$H(z)$		$\{\alpha := x, \beta := y\}$	

(b) The evolution of the marking without token consumption. By not consuming the token $\{\alpha := x\}$ in p_0 at t_{i+1}, the model now considers both the possible assignments $\beta := y$ and $\beta := z$ in p_1 and can proceed at t_{i+3} in producing tokens in p_2.

		Markings		
Time	**Event**	$M(p_0)$	$M(p_1)$	$M(p_2)$
t_i	$F(x)$	$\{\alpha := x\}$		
t_{i+1}	$G(y)$	$\{\alpha := x\}$	$\{\alpha := x, \beta := y\}$	
t_{i+2}	$G(z)$	$\{\alpha := x\}$	$\{\alpha := x, \beta := y\}, \{\alpha := x, \beta := z\}$	
t_{i+3}	$H(z)$	$\{\alpha := x\}$	$\{\alpha := x, \beta := y\}, \{\alpha := x, \beta := z\}$	$\{\alpha := x, \beta := z\}$

In contrast to Petri Nets, all possible combinations of tokens are considered at once. The reason for this is because upon consuming the event we do not know yet which combination of input tokens will eventually lead to a token in an accepting place. Choosing only a single token arbitrarily might result in greedily choosing the wrong token, making the net not progress further. Therefore, to allow for multiple paths to be explored, a Behavior Net considers each combination of tokens during a transition instead.

3.6 Token Consumption

Another crucial aspect of Behavior Nets is that input tokens of a transition are interpreted but no longer consumed upon transitioning. Once a token is produced and put in a place, it always remains in that place and is never 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 4, and a sequence of events that contains the sub-sequence $(F(x), G(y), G(z), H(z))$. 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 problem is demonstrated in Table 3(a). Upon processing the first call to g, a net following the standard execution rules 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 call to g, the transition between p_0 and p_1 would no longer be enabled since no token would be present anymore at p_0. This causes the model to get stuck with a token that (incorrectly) assigns $\beta := y$, and the option to assign $\beta = z$ will never be

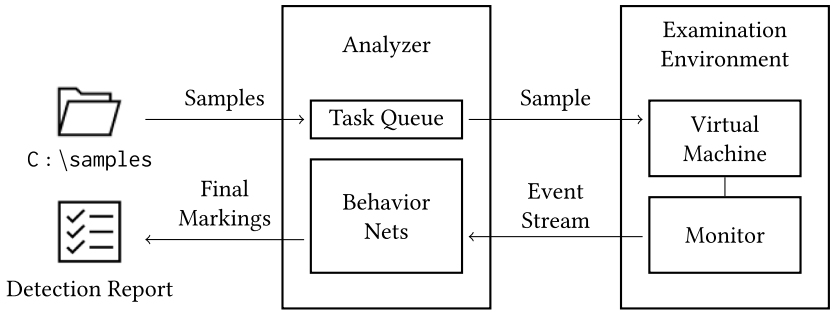


Fig. 5. Framework architecture overview.

considered. However, if we preserve the token at p_0, then both options will be considered at the second g call, and as such the model can continue progressing, as shown in Table 3(b).

A downside of not consuming tokens is that it can potentially lead to *overflowing* the net with tokens. However, since Behavior Nets are meant to model dependency relations and thus cannot contain cycles, this is only a theoretical issue that would not be a problem in practice for the typical use-case of analyzing an event stream produced by a sandbox. Furthermore, we discard any duplicated tokens present at a single place. This is an acceptable change, as semantically equivalent tokens do not provide any new contextual information about a potential final matching of symbolic variables to their concrete values.

4 Framework Architecture

Leveraging Behavior Nets, we build a framework that can automatically characterize the behavior of malware samples based on a set of Behavior Nets. Figure 5 depicts an overview of our framework, consisting of two components. The *Analyzer* acts as a front-end and is implemented in ~5,600 lines of C# code, excluding unit tests (~2,200 lines). It takes samples as input and uploads them to an isolated *Examination Environment*. The Examination Environment executes each sample in a **Virtual Machine (VM)** and records an API call trace which is sent back to the Analyzer. The Analyzer then runs this trace through our Behavior Nets, and reports back which of the behaviors were recognized.

4.1 The Analyzer

The Analyzer maintains and evaluates all the Behavior Nets that need to be considered when analyzing the produced event streams. Such Behavior Nets are built from our repository of behavior specifications. For this, we designed and built a **Domain Specific Language (DSL)** that allows for defining graph structures and event patterns. Our DSL is heavily inspired by the GraphViz DOT language [9] to specify graph-like structures, and YARA [10] and Haskell [6] for their pattern-matching capabilities. While the main focus of this research is on the characterization and use of the different code injection techniques, having a DSL readily available in our reference implementation makes the analyzer easily extensible to include other types of behavior. Furthermore, new types of code injection might be discovered in the future which could then be added on demand as well.

An example of a Behavior Net expressed in our DSL implementing the APC Shell Injection technique can be found in Listing 1. The snippet starts with the declaration of the four places p_0, p_1, p_2, p_3 as well as an additional accepting place p_4. Then follows a collection of transition declarations, each defining an event pattern. Note how the symbolic variables are introduced in the parameters of each event, the use of the discard symbol (‘_’) to ignore irrelevant parameters,

and that extra constraints added to the transition are added in an optional *where* clause. Finally, we can see the edges being drawn between the places and transitions at the bottom, effectively constructing the graph as depicted in Figure 2.

```
behavior "APC Shell Injection" {
  place [p0 p1 p2 p3]
  place p4 accepting

  transition t0 {
    NtOpenProcess(processHandle, _, _, _)
  }
  transition t1 {
    NtAllocateVirtualMemory(processHandle, allocAddress, _, allocSize, _, _)
  }
  transition t2 {
    NtWriteVirtualMemory(processHandle, writeAddress, _, _, _)
    where
      writeAddress in [allocAddress..(allocAddress + allocSize)]
  }
  transition t3 {
    NtOpenThread(threadHandle, _, _, _)
  }
  transition t4 {
    NtQueueApcThread(threadHandle, _, startAddress, _, _)
    where
      startAddress in [allocAddress..(allocAddress + allocSize)]
  }

  t0 -> p0 -> t1 -> p1 -> t2 -> p2 -> t4
  t3 -> p3 -> t4
  t4 -> p4
}
```

Listing 1. A Behavior Net expressed using our Domain Specific Language (DSL), modeling the APC Shell Injection technique. This net is equivalent to the net depicted in Figure 2.

4.2 Examination Environment

To be able to perform dynamic analysis, we rely on running samples in an isolated execution environment. Our reference implementation is built on top of DRAKVUF [33]. DRAKVUF is a virtualization-based, agentless, black-box binary analysis system that allows for monitoring API calls, system calls, network traffic, and file system events.

There are several reasons why DRAKVUF is an ideal tool for a monitoring system. First, in comparison to other popular solutions (e.g., Cuckoo [5]), DRAKVUF can monitor an entire system as opposed to just individual processes. Given the nature of code injection techniques, this is a crucial requirement for us. Furthermore, DRAKVUF observes runtime activity from outside of the VM itself, by interfacing directly into the underlying virtualization software. This means that it does not require an agent within the VM to implement the instrumentation. Consequently, this vastly reduces the risk of being fingerprinted by evasive samples.

We use Windows 10 as an operating system for the VM since it is the most market-dominant OS at the time of conducting this research [57]. To remove 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. In fact, these services might unnecessarily prevent the samples from running, or introduce artifacts in the streams as a result of their own use of code injection to perform their own monitoring.

We configured our analysis environment to allow access to the Internet, since malware often relies on a connection to a remote control server, or it checks connectivity as a means of detecting analysis environments. However, following the community practices [51, 52], we enforced limited connectivity. We let the malware run limited time, deny potentially harmful traffic (e.g., spam), and deploy our system on a separate sub-network where no production machines are connected.

Finally, after each analysis, we roll back the VM to a clean snapshot to revert any side effects that malware might introduce. This also prevents potential denial of service attempts. These countermeasures were approved by the Ethics Committee of our institution.

5 Experimental Results

We now continue with testing our reference implementation of Behavior Nets to assess their capabilities of characterizing the different types of code injection techniques. We also perform a large-scale measurement of the general prevalence of the various code injection techniques in the current malware scene.

5.1 Datasets and Setup

Ground Truth Dataset. To verify that our behavior characterization framework using Behavior Nets correctly classifies the studied code injection techniques, we first assembled a *ground truth* dataset of 63 code injection samples covering all the studied techniques, averaging 3.7 samples per injection technique. Our dataset contains both samples that we implemented ourselves, as well as handpicked open-source implementations and real-world samples which were all manually verified. We also include 20 malicious samples that do not adopt code injection, as well as 1,147 benign applications to test against event streams that contain only benign behavioral data. The benign samples include 976 executables from `C:\Windows\System32` and `C:\Windows\SysWOW64`, as well as 171 popular applications, e.g., VLC Media Player and WinSCP. We used the portable versions of these popular applications to avoid needing to interact with any installation wizards or similar, and thus make the pipeline easier to implement.

Real-world Malware Dataset. Next, to assess that Behavior Nets can also be used on scale with real-world samples for which we do not have their original source code available, we also did a measurement study on the general prevalence of code injection techniques in the wild. We collected 47,128 random samples from the VirusTotal Academic Datasets [58] spread over the years 2017–2021 and ensured each sample was flagged by at least three AV engines (as suggested by related work [61]). We then used AVClass [54] to assign samples to family labels. Table 4 describes this resulting dataset and its family distribution.

Analysis Timeout. According to previous work [32], around 65% of malware runs completely in less than 2, and 81% does not need longer than 10 minutes to fully cover its entire state space. Since the main use-case of code injection is to be an evasion technique, it is likely also one of the first actions the malware performs. Therefore, we pick 6 minutes as a time limit per sample for our prevalence measurement.

5.2 Framework Assessment

Table 5 shows an overview of the classification capabilities of our framework using Behavior Nets on our ground truth dataset. We make a distinction between picking up on the presence of code injection and exactly classifying the techniques. In the following, we will discuss the performance of our framework in more detail.

5.2.1 Classification Capabilities. Our framework successfully recognizes the usage of code injection for all techniques using Behavior Nets, except for *IAT Hooking*. While this technique is

Table 4. Malware Family Distribution in Our Dataset

	Total		2017		2018		2019		2020		2021	
Family	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**	**Cnt.**	**Pos.**
virlock	5,783	0.5%	111	1.8%	131	0.8%	301	9.3%	5,057	0.0%	183	0.0%
dinwod	3,180	0.1%	2,763	0.0%	71	2.8%	71	0.0%	72	0.0%	203	0.0%
sivis	1,066	0.0%	12	0.0%	86	0.0%	71	0.0%	19	0.0%	878	0.0%
berbew	862	99.0%	144	100.0%	12	100.0%	283	100.0%	105	96.2%	318	98.4%
upatre	862	0.2%	190	1.1%	209	0.0%	187	0.0%	189	0.0%	87	0.0%
virut	861	1.4%	200	5.0%	138	0.0%	486	0.2%	33	3.0%	4	0.0%
delf	843	5.8%	31	3.2%	52	3.9%	189	13.2%	136	15.4%	435	0.0%
kolabc	837	0.0%	2	0.0%	12	0.0%	8	0.0%	0	0.0%	815	0.0%
vobfus	816	1.2%	156	0.6%	225	1.3%	41	9.8%	14	0.0%	380	0.5%
wapomi	738	0.4%	318	0.9%	63	0.0%	17	0.0%	339	0.0%	1	0.0%
wabot	596	0.0%	377	0.0%	50	0.0%	117	0.0%	43	0.0%	9	0.0%
vindor	594	0.0%	32	0.0%	36	0.0%	37	0.0%	0	0.0%	489	0.0%
allaple	567	0.2%	193	0.5%	88	0.0%	276	0.0%	9	0.0%	1	0.0%
gator	530	0.0%	63	0.0%	2	0.0%	103	0.0%	34	0.0%	328	0.0%
hematite	470	0.0%	15	0.0%	197	0.0%	230	0.0%	26	0.0%	2	0.0%
vtflooder	462	0.4%	137	1.5%	32	0.0%	58	0.0%	23	0.0%	212	0.0%
shipup	428	88.8%	58	94.8%	251	90.0%	59	86.4%	55	81.8%	5	60.0%
gepys	418	89.2%	27	88.9%	277	89.2%	67	82.1%	40	100.0%	7	100.0%
Other	27,215	9.4%	5395	11.1%	6259	9.5%	6424	12.2%	3498	8.3%	5,642	4.9%
Total	47,128	9.1%	10,224	8.3%	8,191	13.4%	9,025	13.6%	9,692	5.1%	9,999	6.1%

The columns indicate the sample count and the fraction of positive samples.

Table 5. Overview of All Recognized Code Injection Techniques

	Behavior Nets		Cuckoo Sandbox	
Technique	**Match**	**Exact**	**Suspect**	**Detect**
Process Hollowing	✓	✓	✓	✓
Thread 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	✓	✓		

Match indicates some form of injection was recognized. *Exact* indicates a correct identification of the technique. *Suspect* and *Detect* indicate a suspicion and a definitive detection respectively as reported by Cuckoo Sandbox [5]. An asterisk (*) indicates it may be confused with another technique.

destructive, we cannot recognize these injections due to Behavior Nets not being able to test for faulty or *absent* behavior as a result of rerouting an API call. Furthermore, this technique only requires two calls to `NtWriteVirtualMemory` for both transmitting and preparing the catalyst, respectively. While we can observe these calls, we cannot distinguish between the ones that place hooks and inject other types of memory. Note that, this does not mean that Behavior Nets are blind to destructive techniques. For example, in *Process Hollowing*, the catalyst always calls `NtSetContextThread` and `NtResumeThread`, whose arguments can be traced back to previously observed transmitter API calls, and thus can be reliably tested for. However, our framework sometimes confuses it with *Thread Hijacking*, as many hollowing implementations are nearly identical to it, and only include an extra call to `NtUnmapViewOfSection` to "hollow" out the victim process before the payload is transmitted. Again, while Behavior Nets can encode this call for Process Hollowing, they cannot encode its absence for Thread Hijacking, causing the latter to be sometimes incorrectly identified as well. Therefore, if both techniques were detected in a sample, we assume that only Process Hollowing was implemented instead.

For three techniques (*PE Injection*, *Reflective DLL Injection*, and *Memory Module Injection*), our framework can recognize the presence of an injection, but not exactly identify the specific technique. The limited granularity of the system call trace causes some techniques to have a near-identical pattern of system calls for their transmitters and catalysts. In this case, the three methods become indistinguishable from *Shellcode Injection*, and can therefore only be classified as such. This is a reasonable compromise, as, since the only difference between these techniques is the format of the actual injected memory, they can be seen as a special case of injected shellcode. Thus, while this classification does not completely reflect the exact exhibited technique, it is not an incorrect classification either. All these techniques belong to the sample class in our taxonomy. We, therefore, refer to this group of injections as *Generic Shell Injection*.

5.2.2 Performance Metrics. All samples that do not implement code injection were correctly marked negative by our evaluation of Behavior Nets. The 1,147 benign Windows applications were also marked negative, except for one System32 program. This program (`osk.exe`) implements an on-screen keyboard and simulates key presses when the user clicks the virtual key buttons. We found that it indeed uses Windows Hook Injection to send the simulated key presses to other processes. This confirms that code injection is also used for legitimate purposes, emphasizing that the use of code injection is insufficient for classifying a sample as malicious.

Our evaluation of Behavior Nets has a true positive rate of 87.50% and an F1-score of 93.0% on the samples that implement code injection. The false negatives are mainly caused by some samples not activating themselves during the analyses. In particular, implementations of *Windows Hook Injection* are susceptible since their catalyst sometimes requires user input (e.g., key presses) to run the payload. Note that, this is not a limitation of Behavior Nets but rather of any examination environment based on dynamic analysis, and could be mitigated by programmatically introducing (random) interactions in the sandbox (as is done in e.g., Cuckoo [5]).

5.3 Importance of Behavior Nets

The core aspect of our Behavior Nets is their ability to model and track event interdependence by imposing symbolic constraints on their arguments. To assess how this aspect improves the classification over existing models that do not consider event context nor the dependency relations between events (e.g., YARA [10], SIGMA [8] or CAPA [3] rules, as well as most Cuckoo [5] signatures), we ran two more experiments on our ground truth data set. First, we remove *all* dependencies in our models, simulating signatures that test for the mere presence of events with some basic heuristics (e.g., testing whether a `NtWriteVirtualMemory` call writes to another

Table 6. True Positive Rate, False Positive Rate, and F1-score of Three Approaches: (1) Testing for the Mere Presence of Relevant APIs; (2) Testing for the Presence and Order of Relevant APIs; (3) Behavior Nets

Approach	Match			Exact		
	TPR	FPR	F1-score	TPR	FPR	F1-score
APIs only	100.00%	57.37%	0.17	89.06%	57.37%	0.15
APIs + order	100.00%	55.43%	0.18	93.75%	55.43%	0.17
Behavior Nets	87.50%	0.00%	**0.93**	87.50%	0.00%	**0.93**

The *Match* and *Exact* columns differentiate between recognizing code injection and precisely classifying the used technique.

process). Second, we reintroduce the general order in which the events should occur but leave out the constraints put on their arguments. Finally, we verify how these augmented models compare to our original Behavior Nets.

Table 6 depicts the confusion matrices when we run the analyses on both the samples that implement code injection, as well as the benign applications that do not. Again, we make the distinction between recognizing the presence of code injection and exactly classifying the used technique, which is crucial when performing an in-depth prevalence measurement (Section 5.5). In both cases, we can see that the false positive rate is significantly higher for the first two approaches. This intuitively makes sense, as an event stream of an entire system contains a lot of noise from background processes. As discussed in Section 3, only testing for the presence or order of events is therefore bound to result in many false positives. Similarly, it is also important to note that the higher true positives for these two approaches are not necessarily actual detections either. Since most techniques rely on very commonly used APIs, a call to such an API will likely be observed regardless of whether an injection occurred or not. Finally, we see a slight drop in the true positive rate achieved by our approach using Behavior Nets. Similar to what was discussed in Section 5.2.2, this can mainly be attributed to the fundamental limitations of dynamic analysis itself, where samples do not always activate themselves.

5.4 Comparison with Existing Tools

We ran our code injection samples through Cuckoo Sandbox [5], a widely used malware analysis tool that also features a large repository of community maintained signatures, including signatures for detecting the use of code injection.[1] The results can be seen in the last two columns of Table 5. In these columns, we see that the majority of samples implementing an active technique were picked up as a *potential* suspect, from which only three were a definitive positive. This is because Cuckoo mainly relies on the presence of single API calls traditionally used by code injections. As stated in Section 3.1 and demonstrated in Section 5.3, considering single APIs is insufficient to test with confidence whether an injection actually occurred, and as such, Cuckoo must resort to marking samples as suspicious. Additionally, we see that most of the passive techniques were not identified by Cuckoo, further confirming our beliefs that passive techniques are often overlooked. In contrast, our Behavior Net-based signatures can detect the interdependence between multiple API calls and do not rely on traditional APIs only. This allows us to tell whether a specific technique was implemented or not more reliably, regardless of whether the technique is active or passive.

[1]Other online sandboxes exists [2, 7] but they are closed-source and require paid subscriptions to access all functionalities (e.g., 64-bit analyses).

Table 7. Observed General Prevalence and Distribution of Code Injection Techniques in the Sample Set from 2017 to 2021

Technique	2017		2018		2019		2020		2021		Total	
Process Hollowing	230	27.2%	260	23.6%	276	22.5%	92	18.5%	92	15.2%	950	22.2%
Thread Hijacking	87	10.3%	123	11.2%	78	6.4%	12	2.4%	52	8.6%	352	8.2%
CTray Hooking	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
APC Shell Injection	2	0.2%	13	1.2%	1	0.1%	2	0.4%	3	0.5%	21	0.5%
APC DLL Injection	0	0.0%	0	0.0%	1	0.1%	1	0.2%	0	0.0%	2	0.1%
Generic Shell Injection	174	20.6%	138	12.6%	218	17.7%	83	16.7%	37	6.1%	650	15.2%
Classic DLL Injection	2	0.2%	4	0.4%	70	5.7%	3	0.6%	0	0.0%	79	1.9%
Shim Injection	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
IFEO	86	10.2%	25	2.3%	61	5.0%	75	15.1%	80	13.2%	327	7.6%
AppInit_DLLs Injection	86	10.2%	519	47.2%	170	13.8%	137	27.5%	19	3.1%	931	21.8%
AppCertDLLs Injection	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
COM Hijacking	240	28.4%	69	6.3%	406	33.0%	111	22.3%	341	56.4%	1167	27.3%
Windows Hook Injection	0	0.0%	4	0.4%	21	1.7%	7	1.4%	0	0.0%	32	0.8%
Total	846	8.3%	1100	13.4%	1229	13.6%	498	5.1%	605	6.1%	4278	9.1%

5.5 Prevalence Measurement

Now that we confirmed our Behavior Nets are capable of distinguishing between different techniques, we can apply them on a larger scale and measure the adoption of the different code injection techniques in the malware scene. Table 4 summarizes the observed prevalence of code injection within our dataset of 47,128 samples. We identified a total of 4,278 samples (9.1%) that perform at least one type of code injection. To further test whether the classifications made by our Behavior Nets are consistent, we picked 20 positive samples covering all the detected techniques and 20 negative samples, and we manually verified our results. To the best of our reversing effort, all the classifications made by our framework were correct. Naturally, this does not exclude the presence of undetected false negatives (which we will discuss in Section 7). Overall, the fraction of samples observed to adopt code injection varies from 5.1% to 13.6% per year. While this fluctuation does not seem to follow any particular motif, the distribution of the implemented techniques over time reveals interesting patterns.

Table 7 and Figure 6(a) show the distribution of the different adopted techniques in our large-scale measurement, and Table 8 shows the generally observed preference of techniques in each of the years 2017–2021. Note that, the percentages do not add up to 100% as some samples implement multiple code injection techniques. Specifically, 94.65% of the positive samples in our dataset manifested one injection technique, 5.26% manifested two techniques, and four samples exhibited three techniques.

We can see that *Process Hollowing* and *Generic Shell Injection* are among the more popular choices of malware authors. Since these are traditional methods, and the majority of malware authors tend to copy code from others [19], this is an expected result and further confirms our Behavior Nets are truthful in characterizing the different variants of code injection correctly. However, interestingly, our Behavior Nets also highlight that the popularity of these two active techniques is decreasing, while other techniques are on a rise. If we aggregate all techniques by their class, as shown in Table 9, we can see that many of these rising techniques are *Configuration-based* injections. Most notably, in 2018, the *AppInit_DLLs Injection* technique almost overcame all active techniques combined on its own, and in 2020, the aggregation of all Configuration-based techniques convincingly surpassed them.

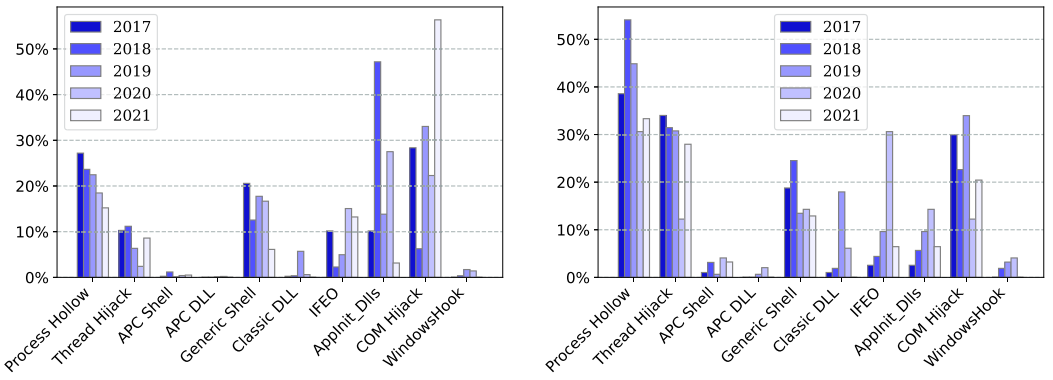
Since samples within a family often employ very similar behaviors [15], and families differ in size, some techniques might be overrepresented in Figure 6. Thus, Figure 6(b) presents a different

Table 8. General Preference of Technique Adopted by Malware Families (*AppInit*: AppInit_DLLs Injection, *COM*: COM Hijacking, *Hollow*: Process Hollowing, *IFEO*: Image File Execution Options, *Shell*: Generic Shellcode Injection, *Thread*: Thread Hijacking, and *WHook*: Windows Hook Injection)

Family	2017	2018	2019	2020	2021	Total
virlock	COM	Shell	Shell			Shell
dinwod	COM	COM				COM
berbew	COM	COM	COM	COM	COM	COM
upatre	COM					COM
virut	IFEO		Shell	WinHook		IFEO
delf	Thread	Thread	Hollow	Hollow		Hollow
vobfus	Hollow	Hollow	Hollow		Hollow	Hollow
wapomi	COM					COM
allaple	COM					COM
vtflooder	COM					COM
shipup	AppInit	AppInit	AppInit	AppInit	AppInit	AppInit
gepys	AppInit	AppInit	AppInit	AppInit	AppInit	AppInit
Other	Hollow	Hollow	Hollow	Shell	Hollow	Hollow
Total	COM	AppInit	COM	AppInit	COM	COM

Table 9. Distribution of Classes of Code Injection Techniques Exhibited by Malware in the Sample Sets from 2017 to 2021

Class	2017		2018		2019		2020		2021		Total	
Active	495	58.5%	538	48.9%	644	52.4%	193	38.8%	184	30.4%	2054	48.0%
Intrusive	319	37.7%	396	36.0%	356	29.0%	107	21.5%	147	24.3%	1325	31.0%
Destructive	317	37.5%	383	34.8%	354	28.8%	104	20.9%	144	23.8%	1302	30.4%
Non-Intrusive	176	20.8%	142	12.9%	288	23.4%	86	17.3%	37	6.1%	729	17.0%
Passive	412	48.7%	617	56.1%	658	53.5%	330	66.3%	440	72.7%	2457	57.4%
Configuration-Based	412	48.7%	613	55.7%	637	51.8%	323	64.9%	440	72.7%	2425	56.7%



(a) Absolute distribution.

(b) Distribution normalized by malware family.

Fig. 6. Observed distribution of code injection techniques exhibited by malware in our dataset.

view of the data, where all samples within the same family are considered as one instead. If one sample within a family performs a given type of code injection, then this family is considered to implement this technique. We see *Process Hollowing* still dominating the market, closely followed by *Thread Hijacking*. We also see that the adoption rates of passive techniques such as *AppInit_DLLs Injection* are reduced, but remain significant and are increasing over the years.

6 Discussion

Our evaluation of Behavior Nets brings valuable insights to the malware research community, which we will discuss in the following.

The Importance of Event Sequences in Signatures. In Sections 2.2 and 3.1 we discussed that a single tactic that malware may implement, including most code injection techniques, often cannot be reduced to a single API alone. Instead, they often comprise multiple APIs, and thus produce multiple events in the event stream, that are invoked in a particular sequence. In Section 5.3, we demonstrated that, without considering this order in which these events are invoked while trying to extract behavioral data (as is often neglected by commonly used state of the art [3, 5, 8]), the accuracy of characterizing malicious behavior drops significantly. Therefore, *behavioral signatures should not just look for the existence of the required events but also take into account the order in which they need to appear.*

The Importance of Context-Aware Signatures. Additionally, as system-wide monitoring is crucial for assessing the total behavior of malware, especially when dealing with code injective malware, malware may still end up using system calls that are indistinguishable from system calls introduced by background processes, even when taking order into account (Table 6). Especially when characterizing passive code injection techniques, which are extra stealthy and shown to be used by a significant portion of the current malware scene (Table 9), this is extra important. Therefore, *considering the context in which a single event is situated is crucial when trying to characterize a sample's behavior,* especially when dealing with system-wide recordings. We demonstrated with Behavior Nets that this can be addressed almost in its entirety if we take the interdependence of these individual events into account, e.g., by correlating their observed arguments and running them through a set of constraints.

Need for Combination of Behavioral Models. As we have seen in Table 5, Behavior Nets cannot recognize the use of the *IAT Hooking* code injection technique. Since this technique does not depend on APIs to activate the injected payload, signatures that look for evidence in an event stream will not find any. Note that, this is a fundamental limitation of any behavioral signature model that actively tries to find evidence for existing abnormal behavior and not the absence of normal behavior. This suggests that *sandbox developers and future researchers should combine multiple approaches to be able to characterize malicious behavior.*

Implications for Future Studies. Our results directly affect future research on malware analysis. Studies based on *dynamic analysis are bound to mischaracterize significant portions of malicious behavior if they do not comprehensively account for the variations in which a specific behavior may manifest as.* Especially when dealing with behavior that comprises multiple APIs that can be reordered without affecting the final outcome, this is crucial. We have also seen that Behavior Nets could be an answer to this, as it does not specify the exact sequence of events but rather considers the general pattern that is expected.

7 Limitations and Future Work

Behavior Nets do not come without their limitations. In the following we will describe the most important ones.

Generalizability of Event Arguments. While the theoretical model of Behavior Net is fairly generalized and allows for virtually any type of argument constraint, it can be difficult to form a set of constraints that are generalized across some classes of malicious behavior. In particular, this is a problem when transitions need to match an event with specific file path-like arguments. Configuration-based code injection techniques are a prime example of this, as they access specific keys in the Registry and thus require matching on arguments with specific Registry key paths. While it is possible to include those paths directly as a constraint in the event pattern, it does not encapsulate the core characteristic of abusing the settings of the OS. If another technique uses a different registry key, a new constraint with this exact key has to be added.

Generalizability of Events. A similar limitation can be found in the use of exact system calls in Behavior Nets. This can be problematic when different sets of API calls result in semantically equivalent behavior. For example, both the `NtCreateFile` and `NtOpenFile` system calls can be used to open a file on the disk for reading. A Behavior Net would then require multiple transition nodes to match both of these options individually. This could be improved by adding a preprocessing phase that lifts specific events in the trace into higher-level *event classes* (e.g., similar to [35]). Alternatively, Behavior Nets could be extended to allow for matching on multiple different types of system events within a single transition node, such that these equivalence classes could be directly built into the graphs themselves. Both options would allow the graph to match a higher abstraction of events that the sandbox observes while avoiding additional structural complexity. We look to explore both of these options in the future.

Correlation versus Causation. The core mechanic behind Behavior Nets is that they can correlate events together based on the similarity of their observed arguments. This works well for event streams spanning a relatively short period, which is what a typical sandbox produces when analyzing a single malware sample dynamically. However, the further apart two events are from each other in time, the confidence that they are related to each other because they share similar-looking arguments decreases. This is because many operating systems, including Windows, implement mechanisms that allow for handles to be reused. For example, when a file handle is closed using `NtClose`, and a new but unrelated file handle is opened using `NtCreateFile`, this new file handle may share the same numerical value as the old handle that was closed before. Since only the raw values are considered and not their origin nor the actual object they reference, a Behavior Net may therefore incorrectly conclude that unrelated events are dependent on each other if these handles were used in its event patterns. Therefore, a Behavior Net may be insufficient when analyzing longer-running event streams where this is more likely to happen. We intend to explore how we can address this problem in the future.

8 Related Work

8.1 Code Injection Identification

Determining the use of code injection has been studied in the past with varied degrees of success. Barabosch et al. proposed a method for detecting code injection leveraging the honeypot paradigm [15], by imitating attractive victim processes and monitoring for anomalies. However, this heavily relies on malware selecting these decoy processes as victims. Furthermore, it also faces the problem of not being able to monitor child processes, rendering many popular techniques such as Process Hollowing undetectable. As an alternative approach, they also proposed to dump the system's memory and search for suspicious memory pages [14]. However, this assumes that benign pages can be distinguished from injected ones, which can be difficult for passive techniques. Furthermore, only some states of a machine are captured, requiring the victim process to be alive upon taking snapshots if we want to find any evidence. Finally, Korczynski and

Yin presented an approach based on system-wide taint analysis to detect the presence of code injection and identify the responsible instructions [31]. Unfortunately, all these approaches do not apply to our measurement framework, as they do not distinguish and classify different injection techniques, which is essential to perform an in-depth study like ours. Proprietary sandboxes, such as ANY.RUN [2] and Joe Sandbox [7], provide indicators of the occurrence of code injection. However, they do not recognize the specific techniques and do not provide information about their analysis approach, as they are fully closed-source.

8.2 Malware Behavior Modeling

Similar to ours, most automated systems for malware behavioral analysis rely on dynamic analysis [2, 4, 5, 7, 33]. While these systems have been very thorough with their examination, they often stop at providing basic interpretations of the logs and leave more advanced conclusions on implemented techniques and tactics up to the analyst. To close this *semantic gap*, various technologies have been developed to help characterize the behavior of malware based on the logs that these types of solutions produce.

Two well-established frameworks that aim to fulfill this task are SIGMA [8] and Mandiant CAPA [3]. Their popularity can mostly be attributed to their standardized methods for specifying rules, making it easy to create new rules or include rules developed by third parties to extend the base rule set. While these rules have proven to be quite effective in describing individual code or event patterns, their core limitations are that they can only specify the existence of relatively high-level system events and cannot be very precise in how individual patterns depend on each other. As was demonstrated in Section 5.3, this is crucial for analyzing logs containing only low-level system events with lots of noise. Especially when trying to detect the use of code injection on this level of abstraction, where ordinary system calls are used a lot, this can be very difficult if not impossible to do reliably without taking event dependency into account.

To be able to define malware behavior more precisely, various graph-based methods have been proposed in the past. In particular, the concept of *malspecs* as described in [18] has been successful in capturing the minimal required events for a behavior to manifest. Similar to Behavior Nets, these are dependency graph-like structures that can precisely describe observed behavior and also support non-determinism. However, malspecs can only describe events using the very specific arguments (exact objects or string literals) that were observed during the examination and thus are hard to generalize for use as behavior signatures. Behavior Nets, on the other hand, support arbitrary expressions for its event pattern constraints, and thus are more flexible and effective in capturing more variations of the same behavior with the same rule. Others such as Martignoni et al. and Kolbitsch et al. have proposed extensions to malspecs in the form of *behavior graphs* [30, 35], which do allow for more complex structures and constraints. However, the evaluation of these graphs heavily relies on taint analysis and thus incurs heavy overhead when applying to a system-wide monitoring solution. Behavior Nets, on the other hand, link events together by correlating the observed values of important arguments the events were produced with, and thus can be evaluated at a much lower computational cost.

Various methods exist to detect and characterize stealthy malware behavior by means of anomaly detection [53, 59]. One limitation of adopting such a strategy is that it requires a database containing a baseline of normal behavior profiles for every potential victim process. In the case of code injection, this task becomes infeasible as operating systems and third-party software installed in the examination environment become more complex. Many potential victim processes (such as `explorer.exe`) are either closed source or too complex to model in a single automaton or graph. Besides, anomalies are at most a weak indicator for code injection, as there are other ways to let benign processes behave abnormally.

9 Conclusion

In this paper, we extended [56] by formalizing and implementing a novel, reusable, context-aware software behavior modeling language called *Behavior Nets*. We showed that Behavior Nets can be used effectively in precisely modeling malicious behaviors, including state-of-the-art code injection techniques, that solely depend on APIs commonly used by many benign applications. By introducing symbolic variables and event argument constraints, we showed that Behavior Net can use the context in which a single event resides to effectively distinguish relevant events from background noise. We evaluated the effectiveness of this approach and experimentally confirmed that introducing context yields better results in finding and characterizing malicious behavior reliably than strategies often employed by other commonly used sandbox solutions. Finally, our research concluded by providing valuable insights on how future malware analysis research based on dynamic analysis should be conducted.

Acknowledgments

We are grateful to Maya Daneva, as well as all reviewers, who have provided us with invaluable insights and feedback. We also thank VirusTotal for granting us access to their academic malware sample datasets.

References

[1] 2013. BPKT - ARM compiler toolchain assembler reference. https://developer.arm.com/documentation/dui0489/c/arm-and-thumb-instructions/miscellaneous-instructions/bkpt
[2] 2024. ANY.RUN - Interactive malware hunting service. https://any.run/
[3] 2024. CAPA - The FLARE team's open-source tool to identify capabilities in executable files. https://github.com/mandiant/capa
[4] 2024. CAPE Sandbox. https://capesandbox.com/
[5] 2024. Cuckoo Sandbox. https://cuckoosandbox.org/
[6] 2024. Haskell language. https://www.haskell.org/
[7] 2024. Joe Sandbox - Deep malware analysis. https://www.joesandbox.com/
[8] 2024. Sigma - Generic signature format for SIEM systems. https://sigmahq.io/
[9] 2024. The DOT language. https://www.graphviz.org/doc/info/lang.html
[10] 2024. YARA - The pattern matching Swiss knife for malware researchers. https://virustotal.github.io/yara/
[11] Hojjat Aghakhani, Fabio Gritti, Francesco Mecca, Martina Lindorfer, Stefano Ortolani, Davide Balzarotti, Giovanni Vigna, and Christopher Kruegel. 2020. When malware is packin' heat; Limits of machine learning classifiers based on static analysis features. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'20)*.
[12] Omar Alrawi, Moses Ike, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua, Taleb Hirani, Brennan Hill, and Brendan Saltaformaggio. 2021. Forecasting malware capabilities from cyber attack memory images. In *Proceedings of the USENIX Security Symposium*.
[13] AVTest. 2021. Malware Statistics & Trends Report. https://www.av-test.org/en/statistics/malware/
[14] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. 2017. Quincy: Detecting host-based code injection attacks in memory dumps. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*.
[15] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. 2014. Bee master: Detecting host-based code injection attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'14)*.
[16] Thomas Barabosch and Elmar Gerhards-Padilla. 2014. Host-based code injection attacks: A popular technique used by malware. In *Proceedings of the International Conference on Malicious and Unwanted Software (MALWARE'14)*.
[17] Marcus Botacin, Paulo Lí–cio de Geus, and André Grégio. 2019. "VANILLA" malware: Vanishing antiviruses by interleaving layers and layers of attacks. *Journal of Computer Virology and Hacking Techniques* 15, 4 (01 Dec. 2019), 233–247.
[18] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 2007. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*.
[19] Fernando de la Cuadra. 2007. The geneology of malware. *Network Security* (2007).

[20] Dejan Lukan. 2013. Using CreateRemoteThread for DLL injection on Windows. https://resources.infosecinstitute.com/topic/using-createremotethread-for-dll-injection-on-windows/
[21] Dejan Lukan. 2013. Using SetWindowsHookEx for DLL injection on Windows. https://resources.infosecinstitute.com/topic/using-setwindowshookex-for-dll-injection-on-windows/
[22] Min Du, Wenjun Hu, and William Hewlett. 2021. AutoCombo: Automatic malware signature generation through combination rule mining. In *Proceedings of the ACM International Conference on Information & Knowledge Management*.
[23] Elastic Security. 2019. Hunting in memory. https://www.elastic.co/blog/hunting-memory
[24] F-Secure. 2018. Hunting for Application Shim databases. https://blog.f-secure.com/hunting-for-application-shim-databases/
[25] Nicolas Falliere, Liam O. Murchu, and Eric Chien. 2011. *W32.Stuxnet Dossier. White paper, Symantec Corp., Security Response* (2011).
[26] Hasherezade. 2018. PE-Sieve. https://github.com/hasherezade/pe-sieve
[27] Intel 2022. *Intel 64 and IA-32 Architectures Software Developer's Manual.* Intel. Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z.
[28] iRed.team. 2020. Import Adress Table (IAT) Hooking. https://www.ired.team/offensive-security/code-injection-process-injection/import-adress-table-iat-hooking
[29] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A large-scale, automated approach to detecting ransomware. In *Proceedings of the USENIX Security Symposium*.
[30] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, Xiao Feng Wang, et al. 2009. Effective and efficient malware detection at the end host.. In *USENIX Security Symposium*, Vol. 4.
[31] David Korczynski and Heng Yin. 2017. Capturing malware propagations with code injections and code-reuse attacks. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*.
[32] Alexander Küchler, Alessandro Mantovani, Yufei Han, Leyla Bilge, and Davide Balzarotti. 2021. Does every second count? Time-based evolution of malware behavior in sandboxes. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'21)*.
[33] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'14)*.
[34] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. 2012. Shadow attacks: Automatically evading system-call-behavior based malware detection. *Journal in Computer Virology* (2012).
[35] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C. Mitchell. 2008. A layered architecture for detecting malicious behaviors. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID'08)*.
[36] Microsoft. 2012. Understanding shims. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-7/dd837644(v=ws.10)
[37] MITRE ATT&CK. 0220. Process injection: Thread execution hijacking. https://attack.mitre.org/techniques/T1055/011/
[38] MITRE ATT&CK. 2020. Process injection: Asynchronous procedure call. https://attack.mitre.org/techniques/T1055/004/
[39] MITRE ATT&CK. 2020. Event triggered execution: AppCert DLLs. https://attack.mitre.org/techniques/T1546/009/
[40] MITRE ATT&CK. 2020. Event triggered execution: AppInit DLLs. https://attack.mitre.org/techniques/T1546/010/
[41] MITRE ATT&CK. 2020. Event triggered execution: Component object model hijacking. https://attack.mitre.org/techniques/T1546/015/
[42] MITRE ATT&CK. 2020. Process injection: Dynamic-link library injection. https://attack.mitre.org/techniques/T1055/001/
[43] MITRE ATT&CK. 2020. Process injection: Extra Window memory injection. https://attack.mitre.org/techniques/T1055/011/
[44] MITRE ATT&CK. 2020. Process injection: Process hollowing. https://attack.mitre.org/techniques/T1055/012/
[45] T. Murata. 1989. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*.
[46] Mohammad N. Olaimat, Mohd Aizaini Maarof, and Bander Ali S. Al-Rimy. 2021. Ransomware anti-analysis and evasion techniques: A survey and research directions. In *Proceedings of the International Cyber Resilience Conference (CRC'21)*.
[47] Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. 2019. D-TIME: Distributed threadless independent malware execution for runtime obfuscation. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT'19)*.
[48] Pieter Arntz. 2015. An introduction to Image File Execution Options. https://blog.malwarebytes.com/101/2015/12/an-introduction-to-image-file-execution-options/
[49] CERT Polska. 2012. More Human than Human - Flame's Code Injection Techniques. https://cert.pl/en/posts/2012/08/more-human-than-human-flames-code-injection-techniques/

[50] Davide Quarta, Federico Salvioni, Andrea Continella, and Stefano Zanero. 2018. Toward systematically exploring antivirus engines. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'18)* (Paris, France).
[51] Dennis Reidsma, Jeroen van der Ham, and Andrea Continella. 2023. Operationalizing cybersecurity research ethics review: From principles and guidelines to practice. In *Proceedings of the International Workshop on Ethics in Computer Security (EthiCS'23)*.
[52] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P'12)*.
[53] Fred B. Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)* 3, 1 (2000), 30–50.
[54] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. AVclass: A tool for massive malware labeling. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'16)*, Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro (Eds.). Cham.
[55] Sevagas. 2014. PE injection explained. https://blog.sevagas.com/PE-injection-explained
[56] Jerre Starink, Marieke Huisman, Andreas Peter, and Andrea Continella. 2023. Understanding and measuring inter-process code injection in Windows malware. In *Proceedings of the International Conference on Security and Privacy in Communication Networks (SecureComm'23)*.
[57] statcounter. 2024. Desktop Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/desktop/worldwide/
[58] VirusTotal. 2021. VirusTotal Malware Academic Dataset. https://www.virustotal.com/
[59] Qi Wang, Wajih Ul Hassan, Ding Li, Kangkook Jee, Xiao Yu, Kexuan Zou, Junghwan Rhee, Zhengzhang Chen, Wei Cheng, Carl A. Gunter, et al. 2020. You are what you do: Hunting stealthy malware via data provenance analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS'20)*.
[60] James Wyke. 2012. *The ZeroAccess Botnet Mining and Fraud for Massive Financial Gain. Sophos Technical Paper* (2012).
[61] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. 2020. Measuring and modeling the label dynamics of online anti-malware engines. In *Proceedings of the USENIX Security Symposium*.

Received 26 July 2024; revised 18 December 2024; accepted 20 March 2025