Automated analysis of system-wide malware propagation

David Korczynski

Linacre College

University of Oxford

A thesis submitted to the
Department of Computer Science of
University of Oxford
for the degree of

Doctor of Philosophy

Trinity Term 2019

Abstract

In contrast to most benign applications, malware infects its host system. It does so via system-wide execution by injecting code into otherwise benign applications, executing via code-reuse attacks, dynamically generating code and much more. These unconventional, albeit perfectly valid, execution paradigms are used for evasion and obfuscation tactics and pose significant problems to automatic malware analysis environments.

In this thesis, we investigate the problem of system-wide malware execution. We focus on building general and precise techniques to analyse malware that execute throughout the entire system. To demonstrate our techniques, we implement them as part of a malware analysis system called Minerva. We use Minerva to perform extensive empirical studies based on synthetic benchmarks that explore corner-case behaviours as well as real-world malware samples collected from the wild.

The core idea behind our techniques is to analyse system-wide malware execution with a bottom-up approach. To this end, we develop a fundamental technique for capturing the system-wide execution trace of a given malware sample that is independent of the techniques malware use to propagate through the system. We then incrementally build abstractions upon this trace to identify code injections, malware droppers, code-reuse attacks, packed malware and more.

In the final part of our thesis, we extend Minerva with several capabilities to perform large-scale studies. We use these features to characterise system-wide malware propagation at large and extract many interesting high-level views on malware based on our precise and general analysis.

Acknowledgements

First and foremost, I thank Bill Roscoe and Colin O'Halloran for inviting me under their wings and supervising me through the creation of this thesis. I much appreciate the solid feedback whenever in need and the freedom I have had to pursue my own ideas in research.

Next, I thank Heng Yin for inviting me to his research lab in Riverside, California. Your critical feedback and extensive experience helped me steer my research in the right direction, and I thoroughly enjoyed working within a dedicated software systems security research lab.

I express sincere gratitude to my viva examiners, Herbert Bos and Kasper Rasmussen. The viva was a pleasure, and your rigorous feedback significantly improved the overall completeness of my thesis.

I thank Cas Cremers, Michael Goldsmith and Andrew Martin for assessing my internal examinations at Oxford. I thank the people from the research community with whom I have discussed malware analysis and that has written to me about my work. Finally, I thank all of my anonymous reviewers through various publication submissions, many of which gave critical feedback and new insights that inevitably shaped my research.

I thank Julien Vanegue for inviting me to Bloomberg in New York City for a summer internship to apply my knowledge of program analysis in an industrial setting.

I thank all of my friends in the Computer Science Department, Linacre and OUAC. Special thanks Arran, Pedro and Talita. I thank Bella for her incredible support, her limitless supply of hindbærsnitter and always being there during my this thesis. Without you, I would never have managed to survive so many years away from home.

Finally, I would like to thank my family. Thank you for the never-ending support and your unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "*Discovery is a child's privilege. I mean the small child, the child who is not afraid to be wrong, to look silly, to not be serious, and to act differently from everyone else.*"
>
> — Alexander Grothendieck, Récoltes et Semailles, 1986.

As malicious software (malware) become greater in numbers, more complex and more evasive, building effective defensive procedures becomes increasingly difficult. Anti-malware companies receive thousands of suspicious samples every day, and each of these needs analysis to determine whether they are malicious or not. The sheer volume makes it impossible to investigate each sample manually, and automated procedures are needed. At the same time, detecting malware becomes more critical as the damage they do is more and more substantial. The need for defensive tools and techniques is higher than ever, and it is imperative to construct these to secure our systems. We need information-gathering tools and techniques to tell us what malicious applications do, how they avoid detection, how they escalate privileges, and so on.

In the majority of cases when a malware sample needs analysis, the only available artefact is the sample in binary format. To analyse the malware, we must, therefore, start from an interpretation of the bytes and build our understanding of the malware incrementally towards higher levels of abstraction. We rely on binary program analysis to do this.

Binary program analysis is attractive because it gives us an explicit view of malware samples from where we can establish what they do. This attraction, however, comes with the drawback that binary program analysis is a tedious task, and the lack of abstractions at the binary level is a challenge to overcome. Therefore, although

binary analysis gives us a starting point, the problem of constructing effective defensive techniques that provide us with an understanding of the malware samples is far from solved.

A core problem when analysing malware is that malware authors build their applications be analysis resistant. For each defensive analysis technique that we develop, malware authors develop an anti-analysis technique. This attack against defensive analysis techniques has led to an arms race between attacker and defender where we continuously aim to outperform the adversary. Fortunately, over the last several decades, this arms race has significantly raised the bar for malware authors to be successful. However, malware authors continue to discover and deploy new anti-analysis techniques, and we must adapt to keep our systems secure.

In recent years, one of the strategies taken by malware authors that have particularly challenged malware analysis systems is system-wide malware propagation. In short, the problem is that malware execute unlike benign applications by exploring unconventional execution paradigms that make it hard to determine the nature of a given malware application. For example, malware injects code into otherwise benign processes, manipulates benign code to execute on its behalf, deploys dynamically generated code, and these kinds of techniques challenge the current ways of automatic malware analysis. In general, there are two central aspects of system-wide propagation that are responsible for this: the operational context of the malware and the combination of many different anti-analysis techniques.

In system-wide propagation, the operational context of malware is the entire system. To illustrate this, consider the propagation strategy deployed by a malware sample from the Tinba family, as shown in Figure 1.1. The malware first injects code into the benign Windows process `winver.exe` and from there further injects code into the benign Windows process `explorer.exe`. The code in `explorer.exe` then scans the processes executing on the system and injects code into every major browser that it finds, and it is only in these processes the malware reveals its true malicious payload. This example is a typical case and shows how malware executes within the entire context of the machine.

The problem with combinations of anti-analysis techniques in system-wide propagation is that for malware analysis systems to be successful, they must have techniques that void all of the anti-analysis techniques. To illustrate an example of such, consider the typical case where a malware sample contains an encrypted payload and also one or more anti-run time checks. There is minimal information available from an initial static analysis of the malware because it hides behind encryption. Instead, to analyse

the malware, we switch to run time analysis because this causes the malware to decrypt its application logic. Unfortunately, the malware detects the run time analysis, and, therefore, only reveals a limited amount of its application logic, and we must return to static analysis to analyse the decrypted application logic.

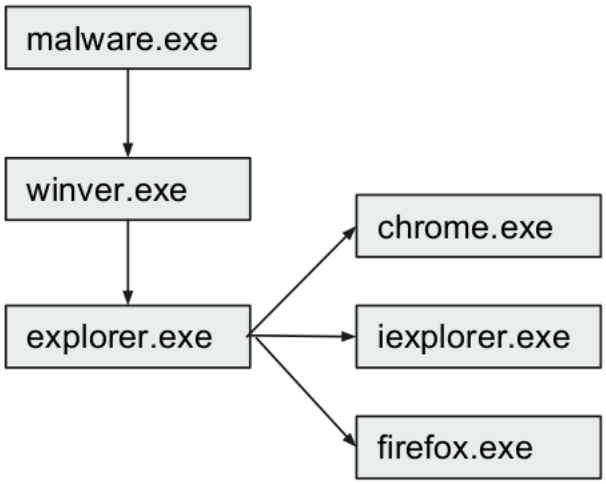


Figure 1.1: Tinba malware host propagation.

To ensure the generality and precision of our defences, we need tools that can automatically analyse malware with system-wide propagation. The goal is to develop proactive solutions, and for this, we need systems that are resistant to entire classes of anti-analysis techniques. In the context of system-wide malware propagation, this means techniques and tools that can analyse malware throughout the entire system and is also capable of handling malware that combines many different anti-analysis techniques. To manage the large volume of samples, it is also desirable that our novel tools and techniques can integrate with current, well-established systems to leverage our existing defensive strengths.

1.1 Scope

In this thesis, the central theme is building practical techniques for automated analysis of malware that use system-wide propagation. We focus on fundamental properties in run time analysis via malware sandboxes and how to leverage the information collected by a sandbox to enhance static analysis. We re-think established malware analysis techniques, question their core assumptions and come up with new solutions that are fundamentally more precise and general.

In practice, we approach the research by analysing malicious samples to investigate how they break current defensive techniques. We identify the invalid assumptions in existing work, redefine those assumptions in a general manner and come up with new solutions that satisfy the redefined assumptions. To validate the correctness of our solutions, we build practical tools and empirically match them with synthetic applications as well as known and unknown malware samples. The motivation for our research problems are, therefore, rooted in real-world malware samples, and we construct our solutions based on generalisation and experimentation.

1.2 Minerva

We present a fully-implemented novel malware analysis sandbox called Minerva. Minerva extends existing sandbox technology in multiple ways and targets analysis of system-wide malware propagation. The primary goal of Minerva is to perform analysis of malware samples based on fundamental principles that are well-defined and formally expressed. Figure 1.2 illustrates the general design of Minerva.

The input to Minerva is a 32-bit binary in the Windows Portable Executable (PE) format, and we execute the binary using full system emulation. Minerva monitors the analysis environment from outside the operating system and currently supports Windows 7, although newer, and older, versions of Windows are straightforward to implement. Minerva traces the execution of the binary using information flow analysis to capture malware execution within the entire system in a general manner. During tracing of the malware, Minerva collects various information about the execution such as code injections, dynamically generated code and code-reuse attacks. The output of the run time analysis then serves as input to a static analysis component that constructs a system-wide disassembly graph of the malware. This system-wide disassembly graph captures intrinsic characteristics of the malware execution that are vital for understanding the inner workings of the malware.

In addition to the core of Minerva itself, we have also built an infrastructure around Minerva that allows the sandbox to be used for large-scale analysis and Minerva builds on top of a record-and-replay framework which makes all analyses of Minerva fully reproducible.

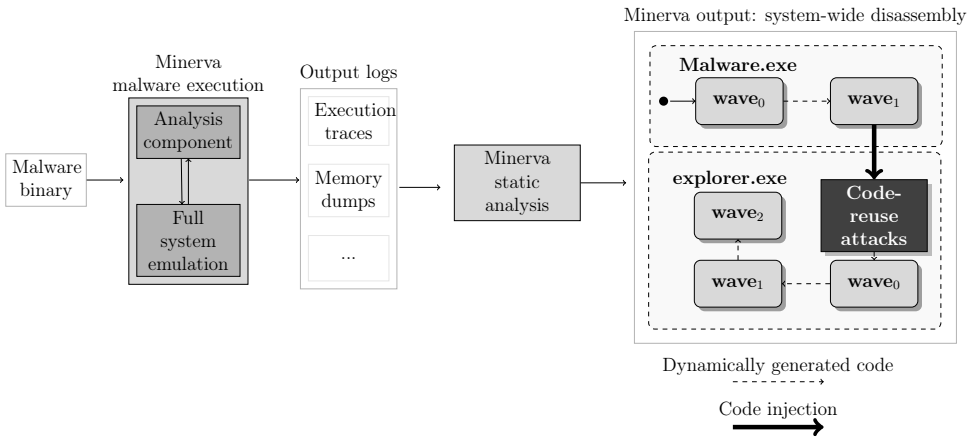


Figure 1.2: Main overview of Minerva workflow.

1.3 Contributions and outline

We extend current malware analysis techniques in several different directions, each of which focus on accurate and general analysis of system-wide malware propagation. Specifically, our contributions are the following:

- **Capturing system-wide malware execution with code injections and code-reuse attacks** (Chapter 4)
  We investigate how to capture a malware execution trace in a system-wide context. This is a fundamental problem that existing literature largely overlooks, and yet most malware sandboxes and analysis techniques implicitly come up with a solution to malware execution tracing. We show that there are significant limitations to existing work when it comes to system-wide propagation and that recent malware samples exploit these weaknesses.

  Theoretically, our result is a novel definition of malware execution trace that is based on information flow and captures malware execution even in the context of system-wide execution and code-reuse attacks. Our work is the first that seriously highlights the problem of dynamically tracing malware execution in a precise and general manner.

  Technically, we design and implement a system based on full system emulation that captures malware propagation using dynamic taint analysis. We evaluate the system extensively against synthetic and real-world malware samples and compare our tool to existing work. Our results show that we outperform existing work by a large margin.

- **Precise system-wide concatic malware unpacking** (Chapter 5)
  We consider the long-standing problem of malware unpacking. Although there is a significant amount of research in this area, existing solutions have fundamental flaws and are limited in their practical architecture. Consequently, they are not well adapted. We present a unified approach that overcomes these limitations and relies on dynamic and static analysis to precisely and generically unpack system-wide malware.

  Theoretically, we formalise the first model of dynamically generated malicious code based on information flow. The main benefit of this is that we precisely capture dynamically generated malicious code independently of who wrote the code but on the basis that it is malicious code. Further to this, we define several algorithms that combine dynamic and static analysis to reproduce PE files based on the captured dynamically generated code.

  Technically, Minerva partitions the malware execution trace into execution waves where each execution wave represents some dynamically generated malicious code, and accurately captures the use of external dependencies in the malware code, even if they are obfuscated. Minerva then performs static analysis with speculative disassembly on each execution wave to output a set of reconstructed PE files with valid import address tables and patched API calls if necessary. The goal is to use sandbox technology to unpack malware and enable follow-up static analysis of the unpacked malware.

- **Characterisation of system-wide propagation in the malware landscape** (Chapter 6)
  We perform a large-scale empirical study to characterise system-wide propagation in the malware landscape. We use the techniques developed throughout this thesis to give new insights into malware from the last seven years. Specifically, we carefully collect a data set of malware samples from the wild, analyse the samples in Minerva and gather vast amounts of statistics about the results. The contributions of our study are threefold: (1) insights about the prevalence and diversity of system-wide malware propagation; (2) insights about the relationship between system-wide propagation and malicious behaviours and (3) insights about the evolution of system-wide malware propagation and insights about inter-family consistency regarding system-wide propagation.

1.4 List of publications

Some of the work in this thesis has been presented and published at various peer-reviewed conferences, where other parts are awaiting publication:

Presented and published.

[94] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1691–1708. ACM, 2017

[91] David Korczynski. Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction. In *IEEE 11th International Conference on Malicious and Unwanted Software, MALWARE 2016, Fajardo, PR, USA, October 18-21, 2016*, pages 31–38. IEEE Computer Society, 2016

Under submission.

[93] David Korczynski. Precise system-wide concatic malware unpacking, 2019.

[92] David Korczynski. A characterisation of system-wide propagation in the malware landscape, 2019.

Chapter 2

Malware sandboxes

> "*When a thing has been said and well said, have no scruple: take it and copy it.*"
>
> — Anatole France, Anatole France en pantoufles by Jean-Jacques Brousson, 1924.

In this chapter, we introduce malware sandboxes in detail. We first present strategies for implementing an underlying environment that enables dynamic analysis (Section 2.1), common levels of analysis granularity (Section 2.2) and two prominent analysis primitives (Section 2.3). These three sections present the core parts that make malware sandboxes general-purpose malware analysis frameworks. We then move on to survey existing sandbox tools proposed and maintained in academia (Section 2.4), concrete analysis tools built on top of these sandboxes that solve specific malware analysis problems (Section 2.5) and discuss how to perform prudent malware experiments (Section 2.6). The chapter ends with related work (Section 2.7).

2.1 Implementation environment

The implementation strategy of a malware sandbox requires crucial design decisions that have a significant impact on the outcome. Besides the usual software development trade-offs, malware sandboxes balance between three main compromises: *visibility, transparency* and *efficiency*. *Visibility* determines how much of the malware execution the sandbox can monitor. *Transparency* is a measurement of how easily the malware can determine it is under analysis. *Efficiency* explains the resources it takes to perform the analysis such as CPU time and memory usage. In this section, we cover the main implementation strategies for malware sandboxes and discuss how each of them balances these three trade-offs.

2.1.1 User- and kernel space analysis

Implementing the analysis directly in user space of the malware execution environment opens up for an efficient sandbox with high visibility. The approach can easily monitor the system and also leverage native application programming interfaces (API) to ease implementation. If the analysis component is implemented in user space only, then the privilege level of the program is important as it will only be able to monitor programs of same or lower execution level. As such, if a malware sample exploits some local service to escalate privileges, it is vital for the analysis to have high enough privileges for continued analysis. Transparency for a user space analysis component is limited because it executes in parallel to the malware and the malware can detect it by merely querying the processes running on the system.

A kernel space analysis component eases some of the limitations of transparency since it can hide its presence from malware that only executes in user space. Furthermore, a kernel component has access to the entire system and can, therefore, gather more information than a user space component, such as the execution of system calls.

A benefit of user- and kernel space analysis is the wide range of deployments. For example, it is possible to deploy user space and kernel space components in both virtual machines, emulated environments and bare-metal environments. However, it is necessary to consider how to extract the analysis results from the analysis system, and this may complicate the implementation. This is because the monitoring component and the malware both execute in the same environment which may disrupt the analysis, such as the malware erasing or encrypting analysis logs.

2.1.2 Emulation-based virtualisation

Virtualisation, in its general sense, is often used as a foundation to implement malware sandboxes. This is because the virtual environments introduce a level of indirection between the environment in which the malware executes (the virtual environment) and the host environment that controls the virtual environment. Emulation is a software-based approach to virtualisation, meaning the entire virtual environment is defined in software. In this way, emulation can use software to expose virtual resources that represent physical devices without needing the physical devices to be present, for example, to enable cross-architecture execution. In this section, we will cover two conventional approaches to creating malware sandboxes using emulation, and then in the following section, we go into detail with emulation's counterpart, namely hardware-assisted virtualisation.

2.1.2.1 Operating system, CPU and memory emulation

Emulating the operating system, CPU and memory allows a sandbox to monitor every aspect of the malware execution and also avoid the malware negatively impacting the host system, e.g. infecting it. The CPU emulator executes the malware in a regular fetch-decode-execute instruction cycle, with the execution of the malware occurring inside the emulated environment. Additionally, system-level services and libraries are emulated, making the execution of the malware tightly controlled by the emulator. This, however, can complicate the implementation of the system as there is a big effort in modelling the environment, including ongoing patches and new versions of the emulated environment. Antivirus solutions often use this type of emulation to deal with new and unseen threats where no signatures exist, especially to decrypt or decompress malware [69].

Emulation has the potential for a high level of transparency because monitoring the malware occurs out-of-the-box. However, to detect the presence of emulation, the malware can rely on inconsistencies between the emulated environment and a real bare-metal environment. For example, if the malware running inside the emulated environment triggers a known CPU bug and the emulated environment does not mimic this CPU bug, then the malware can determine it is running in an emulated environment [128]. This concept applies, of course, to other areas as well, e.g. inconsistencies in OS emulation.

2.1.2.2 Full System Emulation

Full system emulation, sometimes called a machine simulator, gives the features of a real computer system by implementing all hardware facilities in software, including CPU, memory, mass storage and peripheral devices [19, 113]. The virtual environment is, therefore, a virtual machine entirely and allows for complete installation of a commodity operating system. We call the OS that runs in the emulated environment the *guest* and the machine running the emulator the *host*. Full-system emulators are usually run as user-space processes and are powerful for malware analysis because they are in full control of the virtual machine. However, the performance overhead of executing all hardware in software comes with a significant performance overhead, and full system emulators can easily occur 4x slowdown from native execution [19]. As such, full system emulation offers a high level of transparency and visibility but has poor efficiency.

The analysis component of a full system emulator can be placed outside the guest environment to achieve a high level of visibility and transparency. However, to infer meaningful analysis, the analysis component must interpret the state of the guest operating system (OS) solely from the emulated hardware. This interpretation is required because the malware analysis relies on high-level abstract concepts rather than the state of the physical devices themselves. This problem is known as the *semantic gap*. In general, there are several ways to close the semantic gap and choosing which high-level concepts to extract from the OS depends on the analysis needs. For example, on x86 architectures, the page table base register (CR3) is unique for each process, and by reading this register, sandboxes can differentiate between processes running on the system. To further refine the structure of each process the sandboxes then rely on information about the layout of a process in memory, and to follow the execution of the guest system the sandbox can read the instruction pointer in every fetch-decode-execute loop.

Although full system emulation offers a high level of transparency, it is certainly not bulletproof. Similar to OS, CPU and memory emulation, the malware can detect the emulated environment by looking for inconsistencies in the emulation [128]. Malware can also exploit the performance overhead of the emulator via timing attacks [148], however, naturally, researchers have responded with efforts into enabling emulation of emulator-resistant malware [84, 90].

2.1.3 Hypervisor-based virtualisation

Hypervisor-based virtualisation is a technique that enables virtual machines to execute directly on the CPU for maximum efficiency and, thereby, overcomes the significant performance penalty of emulators. In simple terms, a hypervisor is a piece of system software that will prepare and control the environment in which one or more virtual machine lives, but then let each virtual machine access most of the CPU directly. This can dramatically increase performance over emulation since most guest instructions execute natively, but the approach also constrains the virtual environment by fixing the CPU. In hypervisor-based virtualisation, the guest machines can, therefore, be considered a copy of the real machine rather than a completely independent machine on top of the host as in emulation [12, 29].

Hypervisor-based virtualisation is an extensive area of research that has been studied for several decades. Popek and Goldberg formalised the relationship between a virtual machine and a hypervisor (they call it virtual machine monitor (VMM)) in 1974 and came up with three properties that are necessary for a software package

to qualify as a hypervisor. First, the *efficiency property* states that all innocuous instructions of the virtual machine must execute directly on the hardware without the interference of the hypervisor. Second, the *resource control property* states it must be impossible for the virtual machine to alter the system resources, i.e. memory available to it, wherein these cases the hypervisor must be invoked. Third, the *equivalence property* states the virtual machine must behave in the same way as if the hypervisor is not present, with the two exceptions of timing and resource availability problems [124].

The properties identified by Popek and Goldberg requires a given computer architecture to satisfy several conditions for it to support hypervisor-based virtualisation. For example, the architecture needs to have layered execution since the hypervisor must execute with higher privileges than the virtual machine, and all innocuous instructions must be trap-able since all such instructions in the guest virtual machines need to be controlled and emulated by the hypervisor. In this way hypervisors also use emulation albeit to a much lesser extent than full-system emulation, and we call this central paradigm to hypervisors *trap-and-emulate*.

The need for trapping all innocuous instructions makes many architectures from the past non-virtualisable, including those from Intel and AMD, since they contained innocuous instructions that execute in silence. In order to support virtualisation via direct execution, early hypervisors had to work around these issues, for example, VMWare workstation 1.0 made use of dynamic binary translation [30], and other approaches modified the guest operating system to support the underlying hardware for virtualisation, so-called para-virtualisation [12, 154]. As a response to the increased interest in hypervisors, Intel and AMD began in 2005-2006 to introduce hardware support for virtualisation and this is now present in most of their modern processors via Intel VT [42] and AMD-V [4], respectively. This support has continued to evolve, and these hardware extensions currently include several virtualisation-specific instructions that allow easy management of the trap-and-emulate paradigm as well as support for MMU virtualisation. The hardware support developed by Intel and AMD has given birth to several hypervisors such as KVM [89] and Microsoft's Hyper-V [43]. Furthermore, this added support from the hardware is also why hypervisor-based virtualisation is often referred to as hardware-assisted virtualisation.

Sandbox technologies use hypervisors because of their significantly better performance over emulation and also because hypervisors are arguably more transparent as the guest executes directly on the CPU. A common sandbox approach that uses hypervisors is to place the analysis component inside the guest machine, as discussed

in Section 2.1.1. Another approach is placing the analysis component directly into the hypervisor, which forces the sandbox to perform its analysis from outside-the-box since the hypervisor only has access to the state of the VM.

There are several differences between emulators and hypervisors in terms of implementing a malware sandbox that sits outside-the-box. This is because monitoring the guest from the host is more cumbersome in hypervisors since the guest is executing directly on the host's hardware. For example, collecting an instruction-level execution trace through a hypervisor is more complicated than intercepting the fetch-decode-execute loop in emulators. In hypervisors, a common approach to obtain an instruction-level trace is to single-step the execution by setting the trap flag in the EFLAGS register [52]. Single-stepping initiates an interrupt for each instruction execution and the hypervisor then catches these interrupts in order to collect the trace. However, the disadvantage of this is a significant performance penalty as single-stepping is expensive. More recently, systems have explored monitoring execution in the guest by manipulating the NX bit of guest memory [49, 157]. For example, CXPInspector does this via intermodular transition monitoring and uses it to capture API and system calls efficiently, purely from the hypervisor. The technique significantly improves the performance over single-stepping, albeit is slightly more coarse granular [157].

Despite the performance and transparency benefits of hypervisors, they are, however, not a silver bullet. Several approaches have been suggested to detect their presence by identifying hypervisor-specific artifacts [62] and by way of timing attacks [24].

2.1.4 Dynamic binary instrumentation

Full system emulation faces significant performance penalties. One of the drawbacks of sandboxes based on full system emulation is that they execute large portions of code within the emulated environment that is irrelevant for the underlying malware analysis. For example, often, it is only necessary to monitor a few of the processes in the guest system and just a few of the instructions within each of these processes. Dynamic binary instrumentation (DBI) provides an alternative to full system emulation by dynamically instrumenting specific processes and have everything else run natively [25, 28, 111, 117].

DBI engines instrument binaries at run time and usually target user space applications. The efficiency of the approach is achieved by deploying a just-in-time (JIT) compiler to instrument and even optimise code. DBI engines usually target a single

application, and due to their instruction-level view, they have detailed visibility of the application running. However, because DBI engines instrument single processes, the analysis component is implemented and run in the same virtual address space as the malware. This limits both the system-wide visibility in case the malware propagates to multiple processes as well as the transparency because it is easy for malware to identify the presence of a DBI engine. Because the instrumentation only happens in one process, there is a significant performance gain compared to emulating the entire machine, and there is no need to close the semantic gap, which eases the implementation effort.

2.1.5 Hybrid approaches

There are significant trade-offs between emulation and hypervisor-based virtualisation. Hypervisors have a better performance where emulation allows a more refined view into the analysis environment, and hypervisors require the same CPU architecture between the guest and the host where emulation does not. There have been several efforts into combining the two with the general idea of reserving emulation for heavy analysis and use hypervisors otherwise.

Ho et al. [73] proposed to combine virtualisation via the Xen hypervisor [12] with full-system emulation via QEMU. They propose a system where the guest executes by way of the hypervisor during normal conditions and then performs emulation on demand, i.e. the virtualised environment can migrate back and forth between hypervisor and full-system emulator. They use the system to demonstrate efficient full-system taint analysis such that heavy taint analysis is only performed during emulation, and a more lightweight analysis is performed when the hypervisor is in control. A similar approach to emulation on demand was proposed in MOSE by Wei et al. [151] using the KVM hypervisor [89] and also the QEMU full-system emulator. Their idea is to migrate hypervisor environments into full system emulation whenever a crash or similar happens in the hypervisor guest, and then use fine-grained analysis via full system emulation to root-cause the fault, try to fix the fault and then return to hypervisor-based virtualisation.

Another hybrid approach is to record an execution using a hypervisor and then replay the execution in an emulated environment. Aftersight first proposed this in order to decouple the dynamic analysis from the actual execution [38]. Aftersight records all non-deterministic input to the guest system via the VMWare Workstation hypervisor and then replays this recording in a full-system environment via QEMU.

Yan et al. [160, 161] also explored this idea of recording via hypervisor and replaying using emulation; however, they use KVM rather than VMWare Workstation.

An interesting hybrid approach for Android is suggested in Paranoid Android [126]. The idea is to run a synchronised replica of a phone on a security server in the cloud and all security analysis, such as dynamic taint analysis, is then done on the security server. This decoupling of execution and analysis is particularly relevant for phones due to the significant resource constraints in the phone's native environment. In contrast to the other hybrid approaches described in this section, Paranoid Android does not record execution from a hypervisor but instead traces the Android environment with a user space tracer comparable to Linux's *ptrace*, and then replays the recording via Android's emulator. A related approach that also suggests using a security-specific replay server and recording on regular machines is DiskDuster [5], which is a system for automated recovery from intrusions. We discuss the details of DiskDuster later in this thesis in Section 4.9.

2.1.6 Network settings

Nowadays, the majority of malware samples depend on some communication with the Internet to carry out their activities. The specific reasons for requiring Internet access varies, but it enables the malware to get important artefacts like configuration data and bot commands. Malware sandboxes need to consider these aspects because Internet access can significantly impact the analysis results. In this section, we will discuss two approaches that sandboxes use to give the malware samples Internet access, and then in Section 2.6, we will go into more detail about general criteria for designing prudent malware experiments.

2.1.6.1 Simulated network settings

Blocking access to the real Internet but providing a simulated Internet allows the malware execution to be self-contained within the sandbox. The simulated Internet usually implements various popular networking services such as HTTP, HTTPS, DNS and file servers. As a result, if the malware relies on resolving the hostname of a Command and Control server (C&C) then the simulated network will allow it to do so and if the network simulation is complete enough the malware will execute properly and reveal its characteristics. However, malware that relies on specific elements from the Internet, e.g. particular bot commands, will likely stay dormant and not reveal their behaviours specific to these commands.

2.1.6.2 Real Internet access

In comparison to a simulated network, a more radical approach is to give the analysis environment either filtered or full access to the Internet. On the one hand, providing full access to the Internet may potentially allow the malware to engage in real malicious activities such as spreading further or participating in spam and DDOS campaigns. On the other hand, full access to the Internet opens the possibility for malware to receive accurate commands from its C&C server, download configuration files and more. However, in cases where the C&C servers are no longer responding and the malware still requires analysis, real Internet access is likely to do worse than a simulated network, where any host will be available at any time. Real Internet access can, therefore, achieve a high level of coverage but also result in limited analysis, whereas network simulation provides a more stable analysis and guaranteed C&C responses.

Filtered access is achieved by allowing access to the Internet but mitigating the effects of maliciously generated traffic by applying intrusion detection tools and limiting the amount of traffic the malware is allowed to send. This keeps the potential for harmful activities caused by the analysis to a minimum, albeit not limited entirely.

2.2 Analysis granularity

In general, we can analyse malware at many levels of granularity. The selection of granularity is mainly a trade-off between precision and completeness of the analysis, versus the cost in resources like performance and memory usage. In this section, we introduce the three levels of granularity that are the most prevalent in dynamic malware analysis and discuss how they balance these trade-offs.

2.2.1 Persistent changes and network capture

At the coarser level of granularity, we base malware analysis on capturing network traffic and persistent changes in the execution environment from before and after malware execution. To monitor persistent changes, we use forensic techniques and derive features such as files created, files deleted and modifications to the Windows registry. This technique gives us a quick overview of the malware from which we can continue more refined analysis. Furthermore, it can be used to quickly identify where the malware places itself on the system, which is highly useful during incident

response. However, this approach only gives a limited view of the malware and does not reveal much about the temporal aspects of the execution.

Time	Source	Destination	Protocol	Length	Info
0.000000000	192.168.0.3	192.168.0.1	DNS	76	Standard query 0x166c A www.msftncsi.com
0.031859264	192.168.0.3	192.168.0.1	DNS	76	Standard query 0x68db AAAA www.msftncsi.com
19.174215429	192.168.0.3	224.0.0.252	LLMNR	64	Standard query 0x9d5a A wpad
19.304511215	192.168.0.3	224.0.0.252	LLMNR	64	Standard query 0x9d5a A wpad
22.302191143	192.168.0.3	192.168.0.1	DNS	88	Standard query 0x93b4 A localburialinsuranceinfo.com
22.320322985	192.168.0.3	192.168.0.1	DNS	88	Standard query 0xa163 AAAA localburialinsuranceinfo.com
22.561811023	192.168.0.3	192.168.0.1	DNS	80	Standard query 0x5de0 A imagescameraclub.com
22.569978074	192.168.0.3	192.168.0.1	DNS	80	Standard query 0x697b AAAA imagescameraclub.com
22.746826975	192.168.0.3	192.168.0.1	DNS	71	Standard query 0x9136 A 19bee88.com
22.755556876	192.168.0.3	192.168.0.1	DNS	71	Standard query 0xcb03 AAAA 19bee88.com
23.214828263	192.168.0.3	192.168.0.1	DNS	72	Standard query 0x0136 A adrive62.com
23.223770778	192.168.0.3	192.168.0.1	DNS	72	Standard query 0xdfcb AAAA adrive62.com
23.384491172	192.168.0.3	192.168.0.1	DNS	84	Standard query 0x5173 A royalsboostersgbball.com
23.393681576	192.168.0.3	192.168.0.1	DNS	84	Standard query 0xffa0 AAAA royalsboostersgbball.com
23.562618702	192.168.0.3	192.168.0.1	DNS	70	Standard query 0x4718 A ks0407.com
23.572204928	192.168.0.3	192.168.0.1	DNS	70	Standard query 0x9bec AAAA ks0407.com
23.768738028	192.168.0.3	192.168.0.1	DNS	85	Standard query 0x0783 A yahoosupportaustralia.com
23.776476026	192.168.0.3	192.168.0.1	DNS	85	Standard query 0xdf37 AAAA yahoosupportaustralia.com
23.980361892	192.168.0.3	192.168.0.1	DNS	85	Standard query 0xef78 A flexiblepestsolutions.com
23.988327795	192.168.0.3	192.168.0.1	DNS	85	Standard query 0x3140 AAAA flexiblepestsolutions.com
24.141612440	192.168.0.3	192.168.0.1	DNS	76	Standard query 0xc093 A alltimefacts.com
24.151628037	192.168.0.3	192.168.0.1	DNS	76	Standard query 0xc945 AAAA alltimefacts.com
24.327433801	192.168.0.3	192.168.0.1	DNS	73	Standard query 0x894e A thegingod.com
24.338300073	192.168.0.3	192.168.0.1	DNS	73	Standard query 0x9d78 AAAA thegingod.com
24.531283026	192.168.0.3	192.168.0.1	DNS	70	Standard query 0xc108 A frc-pr.com
24.539142567	192.168.0.3	192.168.0.1	DNS	70	Standard query 0xf7ee AAAA frc-pr.com
24.723244665	192.168.0.3	192.168.0.1	DNS	81	Standard query 0xcfd1 A gerberinsreferral.com
24.749465674	192.168.0.3	192.168.0.1	DNS	81	Standard query 0x03fc AAAA gerberinsreferral.com

Figure 2.1: Network traffic captured during CryptoWall malware sample execution.

To illustrate the use of capturing network traffic, consider Figure 2.1, which shows a subset of the DNS traffic collected during execution of a CryptoWall sample[1]. The first column from the left shows the amount of seconds into the execution that traffic is observed and the right-most column displays details about the given DNS packet. After about 22.0 seconds of execution time, there is a large portion of hostnames resolved in a short timeframe. From an analysts point of view, knowing that the sandbox executed a malicious sample hints that the resolved hostnames are Command and Control (C&C) servers. The analyst can, therefore, quickly update network intrusion detection systems with blocked DNS names to protect from this sample despite minimal information about the malware sample itself.

The benefit of using a coarse-grained technique like this is an excellent performance and also a high level of transparency. Coarse-grained analysis can also be carried out directly on bare-metal, meaning the analysis can have the same transparency-conditions as most of the victim machines. However, these techniques give little insight about the malware functionality, misses temporary effects and because the operating system has many applications running, including itself, there is not substantial evidence that the malware is indeed responsible for any of the persistent

[1]MD5 sum of the sample e28a0ed74e78e75710b0d46742e407e3

changes that happen to the system. This type of analysis is, therefore, often followed by more detailed analyses to incrementally build up an understanding of the malware.

2.2.2 Function call

In programming, the concept of a function is a key feature in making code reusable and easy to maintain. However, in program analysis, we often abstract functions into higher-level semantics to avoid cumbersome analysis of irrelevant details. For example, consider the function `InternetConnect` provided by the Windows API. From a program analysis perspective, we are only interested in the fact that this function opens an FTP or HTTP session based on various parameters, e.g. hostname, and returns a handle to the established session. We are not interested in the underlying details of how it does that, as it makes no difference to our analysis. In this manner, function call and system call abstraction is an essential granularity in malware analysis. By intercepting function calls in the execution of a program, we can obtain a clear understanding of the program that is often enough to conclude whether it is malicious or not [40, 56, 109, 137]. Furthermore, this level of granularity can give an analyst a clear view of the internals of the malware and use this for precise guidelines in case further manual analysis is needed.

A common technique for intercepting function calls is *hooking*, and there are many different ways to implement this technique. Figure 2.2 displays a basic approach to function hooking that leverages the use of a *trampoline*. To implement the hook, an instruction (`jmp`) that transfers execution to an analysis function (`FuncAHook`) replaces the first set of instructions in the original function. The analysis function contains a wrapped call to a trampoline function (`FuncATrampoline`) and also analysis code for before and after the wrapped function call happens. The trampoline function contains the overwritten instructions of the original function and a call to an offset inside the original function. With this approach, we can place some code in the analysis function to log whenever the hooked-function executes. It is noteworthy that this type of hooking also allows altering the control-flow or changing the parameters that were intended for the original function. Modifying the parameters is, for example, used in cases where malware calls the `sleep` function to delay execution for evasive purposes and the sandbox then changes the parameter to `sleep` that determines how long time the program execution should pause to zero, negating the anti-sandbox technique.

Address	Instruction
; *Function A*	
0x1000	`push ebp`
0x1001	`mov ebp, esp`
0x1003	`sub esp, 0x40`
0x1006	`mov edx, [ebp+0x8]`
0x1009	`mov edx, [ebp+0x0c]`

Address	Instruction
; *Function A*	
0x1000	`jmp FuncAHook`
0x1005	`nop`
0x1006	`mov edx, [ebp+0x8]`
0x1009	`mov edx, [ebp+0x0c]`
...	
; *FuncAHook*	
0x2000	`push ebp`
0x2001	`mov ebp, esp`
...	*Pre-call analysis*
...	`call FuncATrampoline`
...	*Post-call analysis*
...	ret
; *FuncATrampoline*	
0x3000	`push ebp`
0x3001	`mov ebp, esp`
0x3003	`sub esp, 0x40`
0x3006	`jmp FuncA + 6`

Figure 2.2: Basic hooking. FunctionA pre-hooking on the left. Hooked FunctionA on the right.

In comparison to monitoring persistent changes in the operating system, function call monitoring gives a far more precise understanding of the malware under execution. The increased precision is because we can see the function calls of the malware and therefore conclude with much higher certainty the malware-specific behaviours, monitor for temporal effects and also establish a sequential order between the observed events. Furthermore, we can do correlation analysis by monitoring the parameters and return values of each function call and from this make further abstractions on the malware execution.

Following the traffic analysis example from above, Table 2.1 displays the `InternetConnectA` function calls intercepted in the malicious process of the same CryptoWall sample. We see that the malware connects to several different sites and is responsible for a large part of the network traffic shown in the packet capture. However, more importantly, we can observe that not all network traffic from the packet capture is part of the traffic generated by the malware sample (such as the packets resolving the DNS address of `www.msftncsi.com`), thus establishing a more refined understanding of the malware.

The implementation effort required for function call analysis is substantially more than monitoring persistent changes and network capture. For example, an often

Function name	Arguments
InternetConnectA	[lpszServerName : **mabawamathare.org**]
InternetConnectA	[lpszServerName : **texmart.in**]
InternetConnectA	[lpszServerName : **abelindia.com**]
InternetConnectA	[lpszServerName : **salamasisters.org**]
InternetConnectA	[lpszServerName : **imagescameraclub.com**]
InternetConnectA	[lpszServerName : **parsimaj.com**]
InternetConnectA	[lpszServerName : **shrisaisales.in**]
InternetConnectA	[lpszServerName : **thegingod.com**]
InternetConnectA	[lpszServerName : **champagneframeofmind.com**]
InternetConnectA	[lpszServerName : **kingalter.com**]
InternetConnectA	[lpszServerName : **19bee88.com**]
InternetConnectA	[lpszServerName : **myshop.lk**]

Table 2.1: List of InternetConnectA calls from CryptoWall execution.

used approach to implement hooking as described above is instrumenting programs in user space and then have some form of logging mechanisms that can transfer the observations made inside of the sandbox to the analysts' host machine.

Depending on the implementation, transparency can also be a problem when intercepting function calls. The hooking technique above requires patching of the original routines, which makes them susceptible to integrity checks by the malware. However, if function call monitoring is performed outside of the box using full system emulation, the malware analysis remains transparent.

It is far more performance-expensive to monitor function calls than persistent changes because we must instrument the execution. However, the exact performance penalty is mainly dictated by the underlying implementation platform, as described in Section 2.1.

2.2.3 Instruction trace

At a fine level of granularity, sandboxes can trace the malware execution at the instruction-level. The instruction trace presents all details of the malware execution and based on this, we can do precise analysis about the internals of the malware, including temporal effects.

Instruction-level tracing is rarely monitored manually by an analyst because the trace is too big. For example, the instruction trace length of the CryptoWall sample above is about 730,580,000 during 25 seconds of execution in a full system emulation environment. Rather than manual inspection, instruction level tracing is attractive because it is a gateway to many sophisticated and powerful analyses that can give

precise and clear insights to the malware internals, and we show two such techniques in the following section.

2.3 Prominent analysis primitives

Sandboxes consist of several independent components that make up their overall architecture. However, each sandbox is often meant to support several analysis techniques of a similar sort, and this impacts the architecture of the sandbox. In the last decade, dynamic taint analysis for information flow tracking and symbolic execution for multi-path exploration have gained significant interest from the research community. Consequently, these techniques have influenced the architecture of several currently popular malware sandboxes, and in this section, we present an overview of the two techniques.

2.3.1 Dynamic taint analysis

Taint tracking is used to track the propagation of data in a given system. This tracking is powerful because it enables a variety of sophisticated analyses such as checking for leaks of sensitive data, automatic deobfuscation and much more [13, 56, 59, 127, 135, 158, 162, 164]. Two main parts make up dynamic taint tracking systems: (1) a shadow memory that keeps track of what part of the system is "tainted" and (2) propagation policies that describe how the tainted memory propagates according to the operations that occur in the analysed system.

2.3.1.1 Shadow memory

The shadow memory is a mapping of data in the analysed system, e.g. memory and registers, to a separate store that maintains taint information about the data. There are three important design decisions for each shadow memory.

First, the precision of the shadow memory, with the two most common being bit-level and byte-level. Bit-level is naturally more precise but also more challenging to implement and requires more RAM consumption during analysis. Byte-level precision is a conservative approximation because all data that needs to be tainted will be tainted, but some data may be unnecessarily tainted.

Second, whether the shadow memory only keeps track of tainted versus non-tainted information or if it also includes meta-data along with tainted memory, called *taint labels*. Taint labels are powerful, but they require significantly more memory to just capturing taint and non-taint policies.

arr	i	arr[i]	x
		$\mathcal{T}$	$\mathcal{T}$
	$\mathcal{T}$		♠
	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$			♠
$\mathcal{T}$		$\mathcal{T}$	$\mathcal{T}$
$\mathcal{T}$	$\mathcal{T}$		♠
$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$	$\mathcal{T}$

Table 2.2: Given `x = arr[i]` what should the taint of `x` be? Blank is non-tainted, $\mathcal{T}$ is tainted and ♠ shows when both taint and non-taint are reasonable picks.

Third, defining which specific parts of the analysed system the shadow memory can maintain information. For example, in modern x86 systems, there are the main memory, general purpose registers, flags, SIMD registers, I/O devices and more. The relevance of each of these depends on the purpose of the system. Early adopters of whole-system dynamic taint tracking [39] only supported main memory and the general purpose registers on an x86 architecture, where other approaches that focused on process-level dynamic binary instrumentation only support tracking of main memory in a given process [119].

2.3.1.2 Propagation policy

The propagation policy describes how to update the shadow memory based on the actions performed by the system under analysis. There are many different policies to chose and deciding which one that works best for a given task is an extensively studied area [39, 47, 142]. In general, every operation performed by the system must trigger a parallel operation on the shadow memory. For example, when a memory copy operation A←B happens within the analysed system, an equivalent operation, A←B is performed in the shadow memory. As such, if B was tainted then after the operation, A becomes tainted as well. In this way, the propagation policy for direct data dependencies is intuitive and straight-forward. However, taint analysis aims to track data propagation and data dependencies and is, therefore, more than the tracking data flow via explicit relationships. Other data dependencies have multiple reasonable policies and deciding which policy to use in these cases is less straightforward [34].

For address-based dependencies such as `x = arr[i]` (or equivalent binary form `mov eax, [edx + ecx]`), the propagation is less trivial. The general question is

```
// tainted x
if (x == 0) {
    v = 0;
} if (x == 1) {
    v = 1;
} ...
```

Figure 2.3: Implicit information flow via control flow dependency.

whether the assignment should result in a tainted `x` if `arr[i]` is not tainted, but at least the base address `arr` or the index `i` is tainted. This specific problem is often referred to as *pointer tainting* and Table 2.2 shows reasonable taint policies for this problem. The main problem of doing pointer tainting, is that it increases the likelihood of taint-explosion, such that the amount taint in the system grows exponentially and the precision of the analysis that depends on the taint significantly decreases. As such, trial-and-error with domain-specific use-cases often determine the final propagation policy.

In the case of implicit information flow dependencies, the propagation may not be visible within a single instruction, and to illustrate this, we use two examples provided by Egele et al. [57]. The first control-flow dependency example is given in Figure 2.3. The value of `v` clearly depends on `x`. There have been efforts to handle these types of cases by monitoring whenever tainted memory is used in control-flow operations and propagating the taint within each assignment in the code block that follows the conditional statement [56, 116]. The second and more sophisticated example of implicit information flow is given in 2.4. In this case `v` is assigned the value of `x` via implicit use of the variables `a` and `b`. For example, if `x` is 0, then `b` is assigned 1 and because `a` remains 0, the value of `v` is assigned 0. In this case, the solution for the above problem will taint `b`, but not subsequently `v` because `a` was never tainted. One solution to successfully taint `v`, is to taint all variables assigned in both the true and false branch of conditions in which tainted memory occurs. As such, both `a` and `b` in the first conditional statement would be tainted. This propagation policy, however, makes the overall analysis much more complicated because it requires analysis of both branches.

2.3.2 Multi-path exploration via symbolic execution

Traditional dynamic malware analysis techniques generate a single execution trace of the sample under analysis. A single execution trace is limited because it only

```
// tainted x
a = 0; b = 0;
if (x == 1) then a = 1 else b = 1;

if (a == 0) then v = 0;
if (b == 0) then v = 1;
```

Figure 2.4: Implicit information flow example.

reveals a small part of the entire malware. Even worse, this limitation is actively exploited by malware samples that identify if they execute in a sandbox environment to direct execution down a path that displays no evidence of malicious intent. To circumvent this limitation, researchers have explored the possibility of sandboxes that do multi-path execution, and one promising technique to do this is symbolic execution [32, 33, 86].

The main idea behind symbolic execution is to treat some of the data within an application as *symbolic*, i.e. having an arbitrary initial value. During symbolic execution, the semantics of the language is then used to apply constraints on the symbolic values, and by solving these constraints, the analysis can verify properties on the symbolic trace. If the constraints are satisfiable the SMT (satisfiability modulo theories) solver generates concrete values that solve the given equation. For example, to explore multiple paths, for each conditional branch B where a symbolic value is present, the symbolic executor verifies if both the `true` and `false` branches are possible by passing the constraints B and $\neg B$ to the solver in two separate queries. For each satisfiable branch, the analysis forks execution and continues down the given path [26, 115, 159]. We call the set of constraints that are attached to a given execution path the *path predicate*·

Consider the code snippet in Figure 2.5 from a hypothetical malware sample. This sample will only trigger its malicious behaviours the 19th hour of 24th December 2018. A traditional sandbox without multi-path exploration will only reveal the malicious activities if analysis occurs this given date and hour. If not, the malware will not expose its malicious activities, and the sandbox is likely to declare it benign. However, on the evening of the 24th, all of the malware samples in the wild will trigger their behaviours simultaneously. In order to overcome this issue, the sandbox can explore multiple paths in the malware by treating all input to the malware, which in this case is the output of `GetLocalTime`, as symbolic and perform symbolic execution to explore the path in which the predicate ($systemtime.wYear = 2018 \wedge systemtime.wMonth =$

```
SYSTEMTIME systemtime;
GetLocalTime(&systemtime);

if (systemtime.wYear == 2018 &&
    systemtime.wMonth == 12 &&
    systemtime.wDay == 24 &&
    systemtime.wHour == 19)
{
        Attack();
}
```

Figure 2.5: Running example of malware behaviour that triggers at a specific date and hour.

$12 \wedge systemtime.wDay = 24 \wedge systemtime.wHour = 19$) is satisfiable.

2.4 Sandbox tools

We have now covered the fundamental design space for malware sandboxes. In this section, we present several sandboxes built and maintained in the community and also used in the majority of academic research in the area. We present the sandboxes in an order similar to Section 2.1 with the user- and kernel space sandboxes first, then emulated environments and end the section with sandboxes based on virtualisation.

2.4.1 CWSandbox

CWSandbox is a user space analyser that leverages API hooking and DLL injection to analyse malware [156]. It is built purely for analysis of Windows 32-bit platforms, and its monitoring component is a dynamically linked library (DLL). The analysis component, therefore, resides inside of the same OS environment as the malware under analysis. To perform analysis, CWSandbox launches the target malware application in suspended mode and injects the monitor DLL into this process. The monitoring DLL then iterates over the modules linked inside the malicious process and goes over the export address table for each of them. When CWSandbox finds a function inside an export address table that is to be hooked, it overwrites the first few instructions of the particular instructions with a hooking mechanism as described in section 2.2.2. CWSandbox monitors calls to `LoadLibrary` and `LoadLibraryEx` to capture whenever modules are dynamically loaded, and then performs the same iteration over the export address table if any functions are to be hooked. Additionally, CWSandbox deploys

hooking mechanisms to capture whenever the analysed process creates new child processes or injects code into a running process, and in such cases will inject the monitor DLL into the child process as well.

The central concept behind CWSandbox is inline hooking, as described in Section 2.2.2. As such, CWSandbox has function-level visibility and also comes with a high throughput in the number of analyses it can perform daily. In initial experiments, CWSandbox analysed more than 500 binaries per day per machine, using machines with 2GHz CPU processing power and 2 GB of ram. However, the transparency of the approach is low because the analysis is present in the same environment as the malware and inline hooking is also trivial to detect. To improve transparency, CWSandbox makes various efforts to hide the presence of the monitoring DLL by hiding system objects such as modules and files related to the analysis.

The development of CWSandbox seems to have ceased, but Cuckoo Sandbox is another well-known and open-source system that deploys similar techniques [51]. Cuckoo Sandbox is an active and well-maintained project that, besides hobbyists and academics, many organisations in the industry use, including large enterprise infrastructures [23].

2.4.2 Minos, TaintBochs and demand emulation

Early work in full-system taint analysis is relevant in the presentation of malware sandboxes because it is a foundation on which many malware sandboxes are built. However, these systems themselves are generally concerned about the detection of attacks rather than the analysis of malware.

Minos aims to detect whenever untrusted data is used in illegitimate ways such as control-flow hijacking [46] and can mostly be considered an exploit detector. Although Minos is designed to be a hardware integration, the prototype is implemented on top of the Bochs emulator [1]. TaintBochs [39] is a system that is designed to understand data lifetime in a full-system environment. Similar to Minos, TaintBochs is also implemented on top of the Bochs emulator, and, interestingly, TaintBochs is in many ways a semi-automated tool. For example, it supports record-and-replay analysis and also has a time-travel feature that allows the user to scroll back and forth in the system execution. This is in many ways, a sophisticated form of debugging and, furthermore, TaintBochs does not limit itself to one specific use-case such as identifying control-flow attacks but allows the user to be more selective. The authors present empirical with TaintBochs on how to analyse a program's use of sensitive data by tainting keystrokes of passwords and alike, and then monitor how the system under analysis propagates

the sensitive data. In this way, the authors were able to identify improper use of sensitive data in prominent applications like Mozilla Firefox and Windows 2000.

In addition to Minos and TaintBochs the work by Ho et al. [73] also suggests full-system taint analysis for protection of systems by monitoring inappropriate use of tainted data. Ho et al. suggest a hybrid emulation and hypervisor approach as presented in Section 2.1.5, and instead of providing a specific suggestion on how to handle the execution of untrusted data, Ho et al. focus on showing the feasibility of taint analysis to identify attacks. As such, the system they develop throws an exception whenever an attack occurs and then leaves it to the user to decide what happens from there.

2.4.3 Argos

Argos is a system for automated intrusion detection [127]. It is based on the QEMU emulator [19] and uses full system taint analysis to detect attacks similar to the systems described above, i.e. inappropriate use of tainted data. It focuses detection on three types of attacks, namely control-flow hijacking, format string and inappropriate use of systems calls. In this way, Argos detects a broad range of attacks using full system dynamic taint analysis. In addition to this, for each detected attack, Argos proceeds to generate signatures usable by intrusion detection systems automatically. It does this by extracting memory and network artefacts related to the exploits, combines these artefacts into a signature using the longest common sub-string and a novel algorithm called critical exploit string detection (CREST). Although Argos is an autonomous system, the authors also connected Argos to a signature generation and deployment system called SweetBait [125], which further refines the signatures generated by Argos.

2.4.4 TEMU and DECAF

TEMU is a malware analysis platform based on full system emulation and is part of the BitBlaze project [143]. The implementation relies on QEMU [19] and provides features such as extraction of OS semantics to overcome the semantic gap, dynamic taint analysis and a plugin architecture.

The OS semantics extractor provides information about processes and modules on the guest system, threads and also symbol information for Portable Executable (PE) files. TEMU extracts information about processes currently executing on the guest Windows system with a kernel driver injected into the guest OS. This driver

provides a callback routine to notify analysis plugins when certain events occur in the guest system, such as process creation/deletion and loading of modules. The symbol extractor can parse PE files and extract information such as exported symbols for each module, which is useful when implementing OS-aware analyses such as hooking based on function names. TEMU straightforwardly does this by iterating data structures of the PE files via the Process Environment Block (PEB), located at known positions in Windows processes.

TEMU supports information flow analysis through full system dynamic taint analysis. The taint analysis has byte-level precision and captures information flow in both physical memory, the CPU registers, hard disks and network interface buffers. TEMU plugins introduce taint to the system, and the developer can specify several interesting taint sources, e.g. memory, keyboard and network interfaces. For taint propagation, TEMU supports common information flow operations such as data movement operations, Direct Memory Access (DMA) and arithmetic operations.

Similar to CWSandbox, the TEMU project seems to have ceased. However, some of the original authors of TEMU have developed an improved version called DECAF [72]. DECAF is a malware analysis platform built on top of QEMU and follows many of the same conceptual design principles as TEMU, but presents a refined and improved implementation. DECAF is based on a newer version of QEMU and offers efficient bit-wise dynamic taint analysis. DECAF's taint propagation is implemented directly into QEMU's intermediate representation TCG (tiny code generator) which makes it fast but also challenging to modify. Similar to TEMU, DECAF provides an extensive plugin infrastructure with various callbacks to the core of the sandbox. One of the main conceptual improvements DECAF has over TEMU is virtual machine introspection (VMI) to capture high-level in-guest abstractions. As such, DECAF does not rely on any kernel module within the guest and performs all of its analysis from outside the box.

2.4.5 TTAnalyze and Anubis

TTAnalyze is a sandbox built on top of QEMU and performs analysis of Windows executables within the Windows XP environment [15]. Shortly after the release of TTAnalyze, the project dynamically grew into a more general analysis framework which changed its name to Anubis [13] and is used in a large body of work, e.g. [14, 90, 103, 108].

Anubis centres its analysis around monitoring native Windows API functions as well as system calls. To achieve this, Anubis monitors the instruction pointer, and

whenever it is the value of a known entry point for a Windows function, it logs the given call. Similar to DECAF, Anubis derives function addresses by iterating the export address table of each DLL loaded in the processes under analysis. Anubis also offers the ability to focus the analysis on a given process by capturing the values of the CR3 register, and, Anubis can follow the execution of multi-process malware by monitoring for system calls that create new processes.

The development of Anubis seems to have ceased. However, interestingly, the project was taken further by the primary authors into a commercial product [101], and an exciting aspect of this transition into commercialisation is that the authors continued development of the sandbox into a hybrid emulation and hypervisor approach [95]. Unfortunately, the details are not public.

2.4.6 S2E

S2E is a general-purpose framework for whole-system dynamic analysis and symbolic execution [37]. The core feature of S2E is efficient symbolic execution of large and complex systems including the entire Windows stack without any need for modelling the environment. The authors of S2E demonstrate these capabilities by implementing S2E plugins to automatically reverse engineer windows kernel drivers using techniques from previous work published by the same authors [36, 37, 99].

The two conceptually novel ideas that allow S2E to scale against complex systems is *selective symbolic execution* and *execution consistency models*. Selective symbolic execution is a technique that enables symbolic execution on specific parts of the target system and otherwise performs concrete execution. This combination of symbolic and concrete execution minimises the amount of symbolic execution that is done during analysis and can give significant performance benefits. Execution consistency models is a technique to make principled trade-offs in precision and accuracy when doing selective symbolic execution.

S2E builds on top of QEMU [19], the KLEE symbolic execution engine [32] and the LLVM toolchain [102]. To leverage whole-system symbolic execution, S2E translates the QEMU intermediate representation into the LLVM representation and the LLVM code is then parsed to KLEE for symbolic execution.

S2E is designed to be a general-purpose framework for whole-system analysis. As such, many malware analysis, vulnerability hunting and performance profiling tools use S2E [60] and even other general-purpose instrumentation frameworks have borrowed code from the implementation [53].

2.4.7 PANDA

PANDA is a platform for architecture-neutral dynamic analysis that also builds on top of QEMU [53]. PANDA is in many ways similar to TEMU and DECAF in that it provides an extensive plugin infrastructure, supports information flow analysis and uses the same underlying emulator. However, the critical difference between PANDA and DECAF is PANDA's facility to record and replay executions. Specifically, PANDA's approach to record an execution using lightweight techniques and then replay this recording for heavy analysis provides a different type of work-flow to TEMU and DECAF since analysis happens on the replay rather than a live execution.

PANDA records an execution by taking a snapshot of the current state in the emulator, including the contents of memory and registers, and then continues execution. During execution, PANDA captures every non-deterministic input to the CPU comprising: (1) data entering on the CPU port input; (2) hardware interrupts and its parameters and (3) data written to RAM during direct memory access from a peripheral device. In order to replay the execution, every such non-deterministic input gets mapped to a trace point, which consists of the value of the program counter, the instruction count since recording began and the implicit loop variable (`ECX` register on x86). PANDA supports both x86 and x64 as well as ARM.

Regarding transparency, PANDA is mostly similar to TEMU, DECAF, TTAnalyze and ANUBIS in that they are all based on QEMU. However, a significant difference is the fact that PANDA's execution recording is much faster because there is no analysis going on during the recording phase. This performance improvement impacts timing attacks as well as consistency in network connections because systems that do the analysis "live" can face timeouts if the analysis is too heavy. In terms of efficiency, recording incurs an overhead of about 2x vanilla QEMU and replaying about 4x.

The main focus of the PANDA project is to enable deep and heavy analysis on the recorded executions. For example, PANDA uses code from the S2E project [37] that allows it to raise the QEMU TCG IR into LLVM IR such that analysis plugins are easy to write via LLVM passes. A lot of the engineering effort in PANDA is, therefore, devoted to creating a flexible and architecture-agnostic platform for tool development. PANDA comes with an extensive information flow analysis plugin that supports taint labels, which allows PANDA to have information flow support for all of the architectures that QEMU supports. PANDA also comes with virtual-machine introspection plugins for capturing high-level operating system concepts for both Windows XP and Windows 7.

The record and replay approach of PANDA is similar in spirit to Aftersight [38], the work of Yan et al. [160, 161] and Paranoid Android [126] mentioned in Section 2.1.5. However, in contrast to these tools PANDA performs both recording and replaying via QEMU's full system emulation which makes the implementation simpler but the recording less efficient. The drawback in efficiency does not have significant implications for PANDA though, as the system is designed for post-mortem analysis and fast full system emulation in the recording phase is good enough for that.

2.4.8 Ether

Ether is a virtualisation-based sandbox that focuses on achieving high levels of transparency [52]. The motivation of Ether is an increase in malware samples that try to detect the presence of a sandbox to divert execution and not reveal their malicious activities. The goal of Ether is, therefore, to implement a platform for malware analysis that does not induce any side-effects that are unconditionally detectable by malware within the analysed system. Ether does this by using hardware virtualisation extensions and therefore performs all of the analysis externally from the target environment itself. The specific implementation of Ether uses Intel VT and the Xen hypervisor [12].

Ether has various features to support malware analysis tasks such as instruction-level tracing, monitoring memory writes and system calls, and also offers the ability to focus the analysis on a given process. To support instruction-level tracing, Ether sets the trap flag for each instruction inside of the target environment and then captures every single debug exception within the hypervisor. However, to maintain transparency, Ether does not pass any of the debug exceptions to the guest and also hides the trap flag inside of the guest by intercepting every instruction that is capable of reading the trap flag, e.g. `PUSHF`. By changing the result of these instructions to show that the trap flag is not set, the guest remains unaware of the analysis environment. To capture memory writes within the target, Ether induces a page fault at every memory write attempted by the target, traps the fault and prevents the fault from reaching the target machine. To monitor for system call execution, Ether leverages the fact that Windows XP system calls use the `SYSENTER` instruction to initiate system calls. Whenever this instruction executes, a jump is made in the target to a predefined address in the kernel. A special register `SYSENTER_EIP_MSR` defines this address that only kernel mode can access. Ether modifies the value of the `SYSENTER_EIP_MSR` register to a value that is guaranteed not to be present in memory, thus ensuring the occurrence of a page fault. Whenever a page fault occurs

at the chosen address, Ether is notified, intercepts the system call and sets the original expected value in `SYSENTER_EIP_MSR`. Finally, to enable the analysis to focus on a single process, Ether leverages the use of the CR3 register, as explained in 2.1.2.2.

2.4.9 Dkrakvuf

Drakvuf is another virtualisation-based sandbox and also builds on top of the Xen hypervisor [104]. The goal of Drakvuf is to create a sandbox that is optimal regarding scalability, fidelity, stealth and isolation. In this context, isolation is a synonym to our use of transparancy, meaning the analysis of the virtual machines must be strongly isolated from the sandbox itself to avoid tampering. To monitor malware execution in the sandbox, apart from system calls, Drakvuf injects breakpoints (`INT3` instructions) into the code in the sandbox. These breakpoints allow Drakvuf to, for example, monitor API calls and kernel activity. An interesting aspect of Drakvuf that is not considered in-depth in other sandboxes is a stealthy execution of the malware samples. Most commonly, malware sandboxes have some dedicated script or similar inside of the guest environment that launches the execution of a given malware sample. However, to increase evasiveness, Drakvuf instead hijacks execution of a benign Windows process in the guest environment and forces it to execute the `CreateProcessA` API call. Drakvuf supports both 32-bit and 64-bit Windows 7 SP1 for the guest environment; however, much of Drakvuf can easily transfer to other operating systems such as the use of breakpoints to monitor execution.

2.5 Malware analysis techniques

The goal of malware sandboxes is to enable analyses that solve specific problems and improve our defensive capabilities against malware. In the previous sections, we have shown the architecture of malware sandboxes and various concrete sandbox examples. In this section, we present some of the techniques that are built on top of these sandboxes that aim to give definitive answers to concrete malware problems.

2.5.1 Behaviour-based malware detection

The most obvious question to answer when faced with a suspicious sample, and the one that anti-malware companies often need to answer first, is whether the sample is malicious or not. Sandbox technology is one of the primary tools for automatically answering this question. The conventional approach abstracts the events logged by

the sandbox into higher-level malicious behaviours by identifying correlations and dependencies between the logged events, and then uses these abstractions as a resource to conclude if the sample is malicious or not. These higher-level abstractions resemble what we perceive as malicious behaviours, e.g. “performs a DDos attack”, “encrypts files on the hard disk", “changes browser security settings” and so on. One of the directions researchers have explored in this context is formal frameworks designed to specify and verify abstractions within traces of events.

Jacob et al. [78, 79] present the Abstract Malicious Behavioural Language (AMBL) to model malicious behaviours in a generic and human-understandable format. Under the hood, AMBL models malicious behaviours as Attribute Grammars, which are context-free grammars extended with semantic rules and attributes. The language follows object-oriented principles and includes types and dependencies between objects. In total, they present four models of malicious behaviours: duplication (self-replication virus style), propagation, residency and over-infection. As a result of the representation via attribute grammar, detecting malicious behaviours in logs from the dynamic analysis is reduced to the problem of parsing the grammars representing malicious behaviours.

Beaucamps et al. propose similarly to Jacob et al. a formal approach to specifying and verifying behaviour traces collected from dynamic analysis [17, 18]. The approach relies on string rewriting to abstract the traces into more general terms and then First-Order Linear Temporal Logic (FOLTL) to specify malicious behaviour. For example, consider the behaviour of sending a ping to a remote server by calling the `socket` function followed by the `sendto` function, without calling the `closesocket` beforehand. In FOLTL this is expressed as

$$\phi = \exists x, y.socket(x, \alpha) \wedge (\neg closesocket(x)\, \mathbf{U}\, sendto(x, \beta, y))) \tag{2.1}$$

where x is the created socket, α is the network protocol used, β is the data sent of the socket and y is the target. They use the Construction and Analysis of Distributed Processes toolbox [65] to perform the actual model checking.

2.5.2 Automatic unpacking

Run time packing is one of the most popular anti-analysis techniques adopted by malware. From a simplistic point of view, the idea behind a packer is to compress or encrypt the malware payload within a proxy program that contains the encrypted or compressed stub as well as a decryption or decompression routine. The malware sample is then distributed via the proxy program. When the proxy program executes,

it will decrypt or decompress the actual payload and transfer execution to the original malicious payload. The consequences of this obfuscation technique are that initial static analysis of the malware will not reveal much about the activities of the malware because it will just be a binary blob within the executable file. Analysts must then manually investigate the decryption or decompression routine and extract the original payload. It is only after obtaining the payload that it becomes possible to analyse the internals of the malware using static analysis techniques.

Researchers have put many efforts into solving the problem of automatically extracting the malicious payload with the help of sandboxes [21, 52, 81, 83, 112, 147]. The general idea is to use the invariant of packers that any code that is decrypted or decompressed at run time must be dynamically written to memory. As such, the general approach is to execute the malware within the sandbox and monitor the memory written by the malware. Any code that is written at run time and later executed is, therefore, a case of dynamically generated code. To automatically unpack a given malware sample the sandboxes, therefore, dump the dynamically generated code, which will then be used later for static analysis techniques or simply manual investigation of the malware internals.

2.5.3 Malware clustering

Anti-malware companies receive hundreds of thousands of suspicious malware samples every day and have millions of samples in their databases. An important part of the defense against malware is to group these malware samples based on similarity. This grouping has various purposes such as aiding the creation of general defense heuristics, analysing trends and analysing attribution correlations. We rely on clustering schemes to create these similarity groupings of large data sets.

Bailey et al. [6] presented the first study on clustering malware samples. The specific goal of their work is to automate the labelling process of malware samples by naming samples based on semantic similarities. To cluster the malware, they use single-linkage hierarchical clustering with a normalised compression distance. They execute each sample in a virtual machine and use the Backtracker [87] system to log names of spawned processes, modified registry keys, changed file names and also network connection attempts. Based on the output, they merge these features into a behaviour-profile which then serves as input to the clustering algorithm. As such, the features they use are rather coarse-grained in that they look at a higher level of abstraction than system-call and function call analysis.

Bayer et al. [13] proposed a clustering approach based on locality sensitive hashing (LSH) in combination with hierarchical clustering. The core idea is to use LSH for grouping a large data set into smaller groups, and then apply hierarchical clustering within the small groups themselves. To extract features from each malware sample, they rely on the ANUBIS sandbox described in Section 2.4.5. In comparison to Bailey et al., Bayer et al. present more detailed features. They extract system-calls and Windows API functions called by the malware and also define dependencies and correlations between these using dynamic taint analysis. They allow the samples to download files from the internet via HTTP and also contact Internet Relay Chat (IRC) servers, and reroute all other traffic to a virtual machine that runs various simulated network services. The features observed from a given malware sample are then abstracted into a behaviour-profile describing the activities of the sample and clustering is then performed on these behaviour profiles.

The work by Bailey et al. and Bayer et al. both rely purely on dynamic analysis to gather data about each malware sample. However, some work combines sandboxes with static analysis techniques to perform hybrid solutions. Hu et al. [75] propose a hybrid solution to the problem of clustering malware. The goal of their approach is to cluster malware samples based on features extracted from the disassembly of the malware samples, but, to overcome the problem of self-modifying code they rely on a solution similar to the ones described in Section 2.5.2.

2.6 Empirical evaluations of malware

Assessing the attributes of malware sandboxes and the techniques we build on top of them is a complex task that involves substantial empirical work. In the sciences in general, experiments have many roles, and construction of clear and correct experiments is fundamental for tasks such as deriving results and explaining phenomena. This equally applies to our subfield of malware analysis, and, therefore, when we conduct experiments with malware, it is crucial to carry out our experiments correctly. There has only been limited attention to this topic, and the most central work is that of Rossow et al. [133] in which the authors propose a set of guidelines and criteria to ensure malware experiments are prudent and of high quality.

Rossow et al. identify four categories that must be carefully considered in malware experiments. First, the selection of *correct datasets*, such as removal of goodware, balanced malware families, caution of interpreting malware activities that blend with benign activities, and so on. Second, clearly explaining the experimental setup in a

transparent manner, including an explanation of sample selection, labelling of malware families, description of network connectivity and system setup. Third, ensuring *realism* of experiments, by, for example, selecting relevant malware samples, exercising caution when generalising from one OS to another OS, and choosing appropriate malware stimuli. Fourth, ensuring *safe* experimental execution by deploying and clearly describing secure containment policies. To put these guidelines into practice, Rossow et al. suggest a particular set of guideline criteria that can be used to direct an experimental setup, and for the convenience of the reader, we show these criteria in Table 2.3.

Throughout this thesis, we present a large body of empirical work and have made significant efforts to provide rigorous empirical evaluations of the ideas we set out. In most cases we explicitly meet the criteria suggested by Rossow et al. For example, in our extensive empirical studies we maintain an equal number of samples from each malware family, we ensure maliciousness of the samples by requiring several anti-malware solutions to detect them as malicious, and we have taken significant efforts to analyse and clarify false positives. However, sometimes we diverge from the suggested setup, and, therefore, in each chapter whenever we perform an empirical evaluation, we thoroughly clarify how we performed the experiment and also match our given experimental setup with the checklist in Table 2.3. These checklists are also found in Appendix B.

There are four areas where we consistently diverge from their suggested setup. Specifically, we include samples in our experiments that are no longer in active campaigns and may have been sinkholed, we use network simulation rather than allowing direct access to the Internet, we do not apply user interaction in the guest system during our analyses, and we only evaluate our techniques with a single OS. In our experience using samples that are no longer active in the wild have not been a problem as we have still observed many malicious behaviours, and we have also had a great experience using network simulation that is purpose-built for malware-analysis instead of using direct Internet access. In part, network simulation is effective because it allows moot samples to connect to any server they like, regardless of the server being down on the real Internet. Thus, in our setting the network simulation has sometimes stimulated more behaviour than real Internet access. The reason we do not apply user interaction is to do the experiments in a vacuum-like setting that is easily reproducible in similar contexts. To our knowledge, there is no standardised or recognised way to supply user interaction and we, therefore, avoid it. We also consider the problem of applying user interaction in a correct experimental way to

Criterion	Importance	Description
Correct data sets		
Removed goodware	●	Removed legitimate binaries from datasets
Avoided overlays	●	Avoided comparison of execution output with real system output
Balanced families	◐	Training datasets balanced in terms of malware families, not individual specimens
Separated datasets	◐	Appropriately separated training and evaluation datasets based on families
Mitigated artifacts/biases	◐	Discussed and if necessary mitigated analysis artifacts or biases
Higher privileges	◐	Performed analysis with higher privileges than the malware
Transparency		
Interpreted FPs	●	Analyzed when and why the evaluation produced false positives
Interpreted FNs	●	Analyzed when and why the evaluation produced false negatives
Interpreted TPs	●	Analyzed the nature/diversity of true positives
Listed malware families	◐	Listed the family names of the malware samples
Identified environment	◐	Named or described the execution environment
Mentioned OS	◐	Mentioned the operating system used during execution analysis
Described naming	◐	Described the methodology of how malware family names were determined
Described sampling	○	Mentioned the malware sample selection mechanism
Listed malware	○	Listed which malware was when analyzed
Described NAT	○	Described whether NAT was used or not
Mentioned trace duration	○	Described for how long malware traces were recorded.
Realism		
Removed moot samples	●	Explicitly removed outdated or sinkholed samples from dataset
Real-world FP exp.	●	Performed real-world evaluation measuring wrong alarms/classifications
Real-world TP exp.	●	Performed real-world evaluation measuring true positives
Used many families	●	Evaluated against a significant number of malware families
Allowed Internet	◐	Allowed Internet access to malware samples
Added user interaction	○	Explicitly employed user interaction to trigger malware behavior
Used multiple OSes	○	Analyzed malware on multiple operating systems
Safety		
Deployed containment	●	Deployed containment policies to mitigate attacks during malware execution

Table 2.3: List of malware experiments guideline criteria proposed by [133] (Table I in their paper). The second column denotes the importance that [133] devotes to this subject: ● is a must, ◐ should be done, ○ nice to have.

be a little-explored area that needs further investigation in itself. Nonetheless, we acknowledge this can be recognised as a limitation but agree with Rossow et al. that this is "nice to have" and not "a must". Finally, we only evaluate our techniques within the Windows 7 environment and we acknowledge this as a limitation.

2.7 Related work

Dynamic malware analysis surveys. There exist numerous surveys and classifications of dynamic malware analysis. Egele et al. provide a comprehensive reference that covers most of the academic literature up until 2013 [57]. Peter Szor gives a broader introduction to practical malware analysis techniques and also numerous descriptions of malware examples from the wild [145]. Bulazei and Yener give a comprehensive overview of evasion attacks against dynamic malware analysis systems for both web, mobile and PC platforms, which is an excellent reference for both dynamic analysis systems and attacks on many of them [31].

Origins of dynamic malware analysis. Malware sandboxing is not a new idea. However, the first mentioning of it as a means for automatically analysing malicious applications is unclear. Leonard Adleman discusses in his 1990 work on abstract theories of computer viruses the possibility of *"quaranteering programs by executing them in isolated and safe environments before the programs would be deployed on machines where they can actually spread and do harm"* [2]. To the knowledge of the author, this is the first proposal of sandboxing as a means for inferring the maliciousness of an application.

Chapter 3

Outline of research

In this chapter, we outline our main research objectives and the problems in focus. We first present our novel malware analysis framework called Minerva, which is the main practical contribution of this thesis. Following this, we present our main conceptual contributions, all of which Minerva implements.

3.1 Minerva malware analysis framework

Minerva is a malware analysis framework based on dynamic analysis that, on a practical level, can automate many everyday malware analysis tasks often done manually by an analyst. The primary focus of Minerva is to improve precision and generality of malware tracing, and then use this as a fundamental building block for precise and complete solutions to many more specific tasks like behaviour profiling, unpacking and feature extraction for data-centric tasks. To this end, the core of Minerva is a fine-grained sandbox that performs instruction-level malware analysis using dynamic taint-analysis in a system-wide context. Minerva then comes with several tools for specific malware analysis tasks that leverage the precise system-wide malware tracing.

Minerva's sandbox builds on top of PANDA [53]. As described in Section 2.4.7, PANDA is a dynamic analysis framework based on full system emulation and utilises a record-and-replay infrastructure. As such, Minerva is capable of analysing system-wide malware, the execution of the malware recording is fast in terms of full-system emulation, and all experiments performed in Minerva are fully reproducible.

The general work-flow of Minerva is to record the execution of a given sample, replay the execution with Minerva's dynamic analysis and then use one of Minerva's several tools to solve the specific task at hand. The raw output of Minerva can also be used for manual investigation by an analyst to provide significant aid in the reverse engineering process.

In addition to analysing individual samples, Minerva can also analyse samples in batches to support large-scale studies. Various scripts control this large-scale analysis and the only input needed is a folder tree that contains the samples that need analysis. Minerva also comes with a tool for doing extensive data extraction on batches analysed by Minerva, which we use for large-scale empirical studies.

Minerva consists of about 13,000 lines of C/C++ and Python code[1]. All of the code on top of PANDA is built in C/C++ and consists of about 7,000 lines of code, and the majority of our tools that process the output of the sandbox are in Python. Most of the code in Minerva's dynamic analysis is on top of PANDA; however, we have had to modify the main taint analysis plugin that comes with PANDA to be less resource intensive. Specifically, PANDA's `taint2` plugin can quickly use 40+ GB of memory, and to limit this, we removed support for taint-labels and made some data structures more simplistic.

3.2 System-wide malware execution tracing

The first objective of this thesis is to investigate the problem of tracing malware execution in a precise and general manner. Theoretically, our goal is to distinguish between benign and malicious code execution in a sandbox environment without relying on specific signatures for behaviours observed in known malware. The goal is to come up with a fundamental approach for tracing the malware execution that is independent of knowledge about contemporary malware. Practically, the problem is, given some instrumentation environment where a malicious application executes, meaning we know the initial malicious application, extract the execution trace of this application. On a fine-grained level, this requires determining for each instruction executed on some system, whether this instruction belongs to the malware or whether it is benign code.

The problem outlined in the paragraph above is relevant for the majority of research on dynamic malware analysis. Indeed, the problem is addressed implicitly by any tool that relies on tracing the malware execution; however, to our knowledge, the problem has never been addressed explicitly in a rigorous manner.

At first sight, tracing the malware execution may seem simple without many complications. However, in recent years, the problem of tracing a malware execution has been seriously challenged and is becoming increasingly relevant for producing precise analysis tools. This is because malware no longer executes purely in one process

[1]Lines of code are measured using Sloccount[152]

but instead infects its host system by, for example, dropping and executing files embedded in their application [100, 132], injecting code into benign Windows processes [11, 77], using otherwise benign code via code-reuse attacks to perform malicious execution [67, 130] and much more. In general, the design space for executing in non-conventional ways is enormous. In these cases distinguishing benign from malicious code execution becomes fuzzy and current approaches use simple techniques like hooking some well-known APIs that allow malware to execute code in other processes. However, when malware executes in these unconventional ways, the current tools and techniques will end up with unreasonable over-approximations or under-approximations of the malicious execution trace.

As such, the first objective of our thesis is to come up with a solution that is fundamentally strong for tracing the execution of a malicious binary. In current terms, this means capturing an exact execution trace of a program that may inject code into other processes, perform exploits that rely on code-reuse attacks, execute dropped files, and much more. We, therefore, consider the problem to be execution tracing of malware in a system-wide setting.

The solution we come up with is dynamic analysis of malware where the tracing of the malware execution relies exclusively on information flow analysis. This is a fundamental and robust approach to tracing the malware because it is independent of a large variety of evasion techniques that malware use, e.g. spreading via files and processes. Fundamentally, our tracing capabilities do not rely on any knowledge of high-level OS-constructs or knowledge of specific tricks deployed by malware. The idea is then to apply more abstractions on top of this fundamental execution trace to give a high-level understanding of how a given malware sample executes. As such, Minerva solves malware analysis with a bottom-up approach, which is in contrast to previous work that comes with a top-down approach by defining the malware execution trace based on knowledge of evasion techniques, OS-constructs, malware tricks and more. The benefit of our approach, then, is Minerva's ability to analyse malware samples with new and unknown evasion tricks in a general and precise manner.

The implementation of the system-wide tracing corresponds to 4500 lines of C++ on top of PANDA and 1500 lines of Python code that manages the analysis process and also does minor post-processing on the run time analysis.

3.3 System-wide unpacking

The second objective of this thesis is to come up with a solution to automatically unpack packed malware. Packing refers to malware that hides its payload as encrypted or compressed memory at compile time to hinder static analysis, and then at run time the malware decrypts or decompresses this memory in order to execute its payload. Packing is a long-standing problem, and reports from major anti-malware companies estimate about 80% of malware samples come packed [69]. Anti-malware researchers have investigated techniques to automatically unpack applications and the majority of this research [52, 75, 81, 83, 91, 112, 147] focus on capturing dynamically generated code. They focus on dynamically generated code because it is an invariant of decryption and decompression since decrypted or decompressed code must be written at run time. The goal of these unpackers is then to extract all of the dynamically generated code within a given malware sample to enable further analysis.

There is an intricate relationship between executing in a system-wide context and packing. Malware that starts execution as a single process and infects the system in order to execute in multiple processes requires dynamically generated code, i.e. memory that is written at run time and then executed. For example, malicious code injection can be considered dynamically generated code across processes. Dropping files and executing them can be considered dynamically generated code via the file system. As such, the invariant that is used to unpack malware automatically is also an essential invariant in system-wide malware execution.

Although automatic unpacking is a well-studied area, existing unpackers come with several conceptual and practical limitations. First, the invariant of dynamically generated code is ubiquitous across the entire operating system, e.g. writing executable files, loading of programs and just-in-time compilation. Current solutions, therefore, cannot simply extend their techniques to system-wide malware. These cases are relevant because malware that propagates through the system often dynamically generates code *via* benign code. Second, automatic unpacking of malware requires more than just capturing dynamically generated code. For example, the dynamically generated code is often position-independent and comes with API obfuscations that need deobfuscation before the unpacker's output can be used effectively. Third, from a practical point of view, the architecture of many existing unpackers are not well-designed, and their implementation is inherently limited, making them ineffective against most malware samples. As a result of these limitations, existing generic unpackers have not been adopted, and unpacking is still very much a manual process.

We come up with a solution to unpacking that combines dynamic and static analysis. First, we use our foundation for system-wide malware tracing to extend the definition of dynamically generated code to a definition of dynamically generated *malicious code*. This allows us to capture system-wide generation of malicious code and solves the first limitation above. Second, we capture both the precise API calls made by the malware and the dynamically generated code involved in the unpacking and then use static analysis to make these elements into well-formed PE files that only contain malware code. These PE files are constructed such that they are well-suited for follow-up analysis, which solves the second problem highlighted above. Finally, because our architecture uses full system emulation and we take as input a malware sample and output several PE files useful for further analysis, e.g. disassembly in IDA Pro, we solve the third problem above.

The implementation of the system-wide unpacking corresponds to 2500 lines of C++ on top of the system-wide analysis, and about 1500 lines of Python for the static analysis.

3.4 Assessing the malware landscape

The third objective of this thesis is to assess the malware landscape at large regarding system-wide malware execution and packing techniques. To do this, we collect an extensive data set of 65 malware families with samples ranging over a seven-year period, analyse them in our sandbox and systematically collect statistics on the results. The study we perform comprehensively characterises system-wide malware propagation as samples in the wild use it. We investigate four main aspects of the samples in our data set: (1) the prevalence of system-wide malware propagation; (2) the diversity of malware propagation strategies; (3) the relationship between malicious actions and system-wide propagation; and (4) the evolution and inter-family consistency of system-wide propagation.

The implementation of the statistics module corresponds to 3000 lines of Python code.

Chapter 4

Capturing system-wide malware execution with code injections and code-reuse attacks

> "*the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.*"
>
> — Edsger W. Djikstra, The Humble Programmer, ACM Turing Lecture, 1972.

In this chapter, we study malware that executes throughout the entire system and also manipulates otherwise benign code to execute on its behalf. We present a unified approach to tracing malware propagation inside the host in the context of code injections and code-reuse attacks and implement our techniques in Minerva. We perform an empirical evaluation of Minerva by matching it with both synthetic applications and real-world malware and also compare Minerva to previous work. Our results show that the techniques of Minerva substantially improve the precision for collecting malware execution traces and that our approach can capture intrinsic characteristics of novel code injection techniques.

4.1 Introduction

Analysis and detection of system-wide malware execution is a problem that has particularly challenged the anti-malware research community. The problem is that malware *integrates* itself to the system on which it is running using stealthy approaches, often motivated by evasion and privilege escalation. This integration is an important property for our defence systems because many of them detect and analyse malware

based on knowledge about propagation strategies, e.g. host-based intrusion prevention systems (HIPS). Two central aspects of malware propagation are code injections and code-reuse attacks.

Code injection is when the malware writes code to another process on the system to have this code execute. This propagation makes the malware execute under the same process as a legitimate application such as a white-listed process that goes undetected by HIPS. In the cases where the malware relies on well-known techniques and our systems already have signatures, the code injections are, naturally, trivially detected. However, recent reports have shown that novel code injection techniques can go unidentified by fine-grained malware analysis environments [110, 130] and also bypass modern-day HIPS [106, 163]. Evasion via system-wide propagation presents a critical challenge because false negatives in both environments allow malware to operate without detection for a potentially long time.

Traditional code injection techniques use API calls such as `MapViewOfFile`, `WriteProcessMemory` and `CreateRemoteThread`, and previous work that aims to trace malware execution rely on hooking these functions to capture whenever multi-process execution occurs [21, 147]. However, recent malware samples from the wild and anti-malware researchers have started to explore code injections that use exploit-like features such as code-reuse attacks [16, 106, 110, 163]. This means, instead of injecting code via `WriteProcessMemory` and `CreateRemoteThread`, malware, for example, writes memory to a system-global buffer and forces the target process to overwrite its stack in order to execute a ROP chain controlled by the malware. In this way, the malware executes in the target process without explicitly writing memory to it [106].

Automatic detection and analysis of code-reuse attacks have recently received much attention from the research community [48, 67, 88, 123]. However, it is difficult to come up with techniques that generically and precisely detect novel code-reuse attacks, and analysis of special cases seems inevitable [48, 66]. For example, a recent demonstration of a new code injection was shown to bypass both HIPS and the many exploit-mitigations deployed by Windows [107] and this particular injection technique was adopted by malware not long after the injection was first published [16] . The problem with the current tools for analysing code-reuse attacks, besides being few in numbers, is that these tools only focus on the injection and, therefore, provide a local and limited view on the entire malware propagation. This makes them well-suited as reverse engineering aids but not for complete and automated analysis of entire malware propagations [67, 88, 123].

In general, code injections and code-reuse attacks are practical examples of the deep theoretical question: *how to distinguish benign code execution from malicious code execution*. Many applications execute on commodity systems, and these applications interact in a variety of different ways, making it challenging to identify the specific execution context and impact of a given application. Malware is actively exploiting this difficulty, and there remains to be found a general and accurate solution to analyse malware propagation with code injections and code-reuse attacks automatically.

In this chapter, we describe a unified approach for automatically capturing and analysing malware propagation with code injections and code-reuse attacks implemented into Minerva. Minerva catches the malware execution based on a novel approach that relies on taint analysis of the *entire* malware code *in combination* with a model of code-reuse attacks. This combination allows Minerva to follow malware execution in the whole OS without relying on any API hooking, and it also gives Minerva the ability to identify code-reuse attacks within the trace. Minerva then goes deeper by also implementing two novel techniques to capture code injections within this execution trace and extract intrinsic aspects about each of them in the shape of a code injection graph.

Our main contributions are as follows.

- We propose algorithms for complete malware tracing based on taint analysis in combination with a model of code-reuse attacks.
- We propose a fine-grained unified approach for automatically identifying code injections and highlight their intrinsic characteristics.
- We present an implementation of a prototype system and perform empirical evaluation on both synthetic applications and real world malware samples on tracing malware execution purely based on information flow analysis.

The chapter is organised as follows. To introduce the problem, we first present an introduction to system-wide malware executions with a motivating example based on a real-world malware sample (Section 4.2). Next, to reason precisely about the problem, we present an abstract framework that allows us to define malware execution tracing inside of malware sandboxes in a clear and formal manner (Section 4.3). We then present our novel approach to tracing instruction-level system-wide malware execution (Section 4.4) and two different strategies for identifying code injections within a system-wide execution trace (Section 4.5). Following that, we outline the

taint implementation in Minerva (Section 4.6) and present an empirical evaluation of correctness based on both synthetic applications and real-world malware (Section 4.7). Finally, we discuss limitations (Section 4.8) and present related work (Section 4.9).

4.2 System-wide malware executions

In this section, we introduce the motivation and background for our work via a running example and go into detail regarding how it uses code injections and code-reuse attacks.

4.2.1 Motivating example - Gapz malware

The running example we use is a real-world malware sample from the Gapz family[1]. Figure 4.1 shows the initial part of the propagation strategy deployed by the sample. The black circle within the `Malware.exe` process indicates the entry point of the malware, solid arrows present control flow and dashed arrows present data flow.

The malware first dynamically generates several waves of code within its process and incrementally transfers execution to these. The decrypted memory then injects code into the legitimate, and already running, Windows process `explorer.exe`. The malware achieves code injection by overwriting a pointer in `explorer.exe` using the function `SetWindowLong` provided by the Windows API and hijacks execution of `explorer.exe` using a call to another Windows function `SendNotifyMessage`. The hijacked `explorer.exe` process transfers execution to a sequence of code-reuse attacks that write shellcode within `explorer.exe` and transfer execution to the shellcode. The shellcode itself will load a memory mapped file and transfer execution to this file.

Our focus in this chapter is on capturing code injections and code-reuse attacks, and we then return to the dynamically generated code in Chapter 5.

4.2.2 Code injections

In the context of malware, code injection is when the malware writes code to another process, called the target, and then has this code execute. Effectively, code injection allows malware to perform malicious activities in the context of the target process. There are many reasons why malware use code injections and two of the most common

[1]md5 of sample = `0ed4a5e1b9b3e374f1f343250f527167`

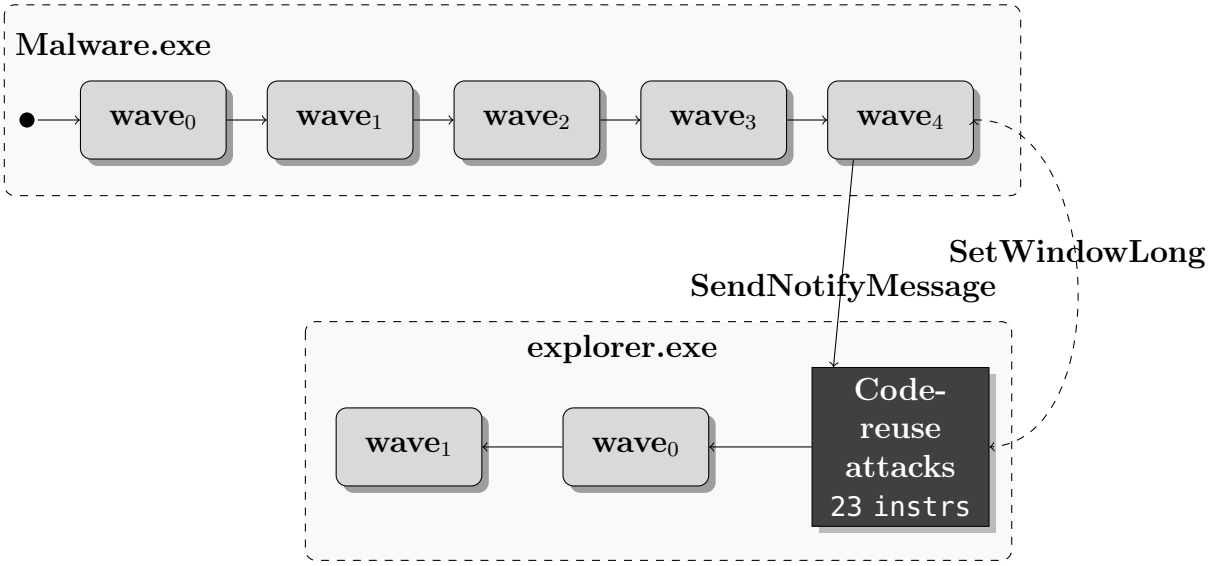


Figure 4.1: Malware propagation of Gapz.

reasons are for evasion purposes and process-level privilege escalation. For example, if the target process is a benign Windows process, then the malicious execution may go undetected because the benign processes are white-listed and not checked by the local HIPS[107].

The motivating example uses a non-traditional case of code injection. The malware avoids `WriteProcessMemory` and `CreateRemoteThread`, or similar API calls, to perform the injection and instead uses two mundane Windows functions, `SendNotifyMessage` and `SetWindowLong`. At the time the malware was first discovered this caused fine-grained malware analysis sandboxes to miss the injection [110].

It is important to clarify that code injections are not necessarily exploits, and many of them are not intended to escalate privileges from an OS perspective. Instead, these are injection techniques that target OS- and application-specific constructs that allow the malware to execute code in another process. As such, there is no strict need for escalating privileges, which makes the number of potential targets plentiful.

4.2.3 Code-reuse attacks

In order to hijack the control flow of an application, malware can manipulate the execution of benign code to perform malware-specific computation even without writing any code to the application itself. We call this type of attack a code-reuse attack. Originally, code-reuse attacks were proposed to enable code execution in the context of non-executable stacks [50]. In this way, the technique requires the ability to hi-

jack execution, e.g. via a memory corruption vulnerability, but rather than executing attacker-written shellcode it relies on reusing blocks of existing code in the victim process and chaining them together in ways that are meaningful for the attacker.

The most popular code-reuse technique is return-oriented programming (ROP), described by Shacham [139]. The basic idea behind ROP is to rely on small code blocks that end in `ret` instructions, called gadgets. Specifically, the attack works by overwriting the stack with addresses of gadgets, such that whenever the `ret` instruction of the first gadget executes it will transfer control to the second gadget, which in turn executes its `ret` instruction and transfers execution to the third gadget, and so on. By combining these gadgets in meaningful ways, the adversary can achieve complex computation and in most real-world cases, even Turing complete computation [27, 35].

Code-reuse attacks are not limited to ROP but can essentially hijack many types of indirect branch instructions. For example, jump-oriented programming (JOP) explores a similar paradigm to ROP but focuses on indirect `jmp` instructions instead of `ret` instructions [20]. Code-reuse attacks can also be obtained by hijacking indirect `call` instructions and, of course, a combination of all three.

Returning to our motivating example, to hijack execution of `explorer.exe` the sample uses the functions `SendNotifyMessage` and `SetWindowLong` to trigger a code-reuse attack in `explorer.exe`. The call to `SendNotifyMessage` triggers a message handler inside `explorer.exe` and a part of the code in this message handler is shown in Figure 4.2. The trigger-point that allows an attacker to achieve arbitrary code execution is that the value returned by `GetWindowLong` (put in the `EAX` register) is a value that is attacker-controlled and can be manipulated from a remote process using `SetWindowLong`. After the call to `GetWindowLong`, `EAX` is copied into `ESI` and a conditional branch is executed. If the condition is false, then execution continues in a basic block where the 4 bytes placed at the address given in `ESI` is copied into `EAX`. Next, three indirect `call` instructions are executed and each of them branch according to the value in `EAX`. Because the malware controls the value returned by `GetWindowLong` it effectively controls the pointer that determines the destination of the three indirect branches. By redirecting these three indirect branch instructions to carefully selected code snippets within `explorer.exe` the malware is able to hijack execution of `explorer.exe` to code that is completely controlled by the malware.

In addition to the malware relying on unorthodox techniques to hijack execution, it is important to highlight that the technique specifically targets `explorer.exe` as it relies on specific code that is unique to `explorer.exe`. Evasive code injection

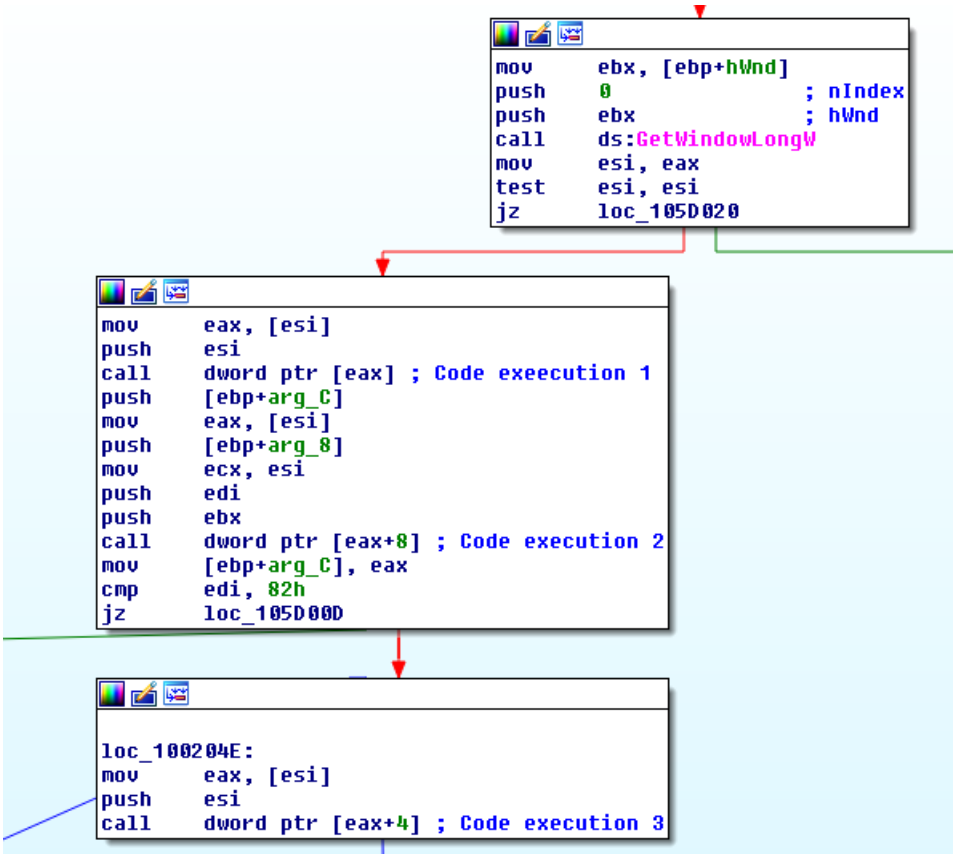


Figure 4.2: Vulnerable message handler in `explorer.exe`

techniques are, therefore, actively researched, and it is difficult to eliminate all of them within an operating system because it requires analysis of each application.

4.3 Abstract model of execution environment

A core goal of this chapter is to develop techniques that capture system-wide malware execution and also identify code injections and code-reuse attacks within the execution. We, therefore, need a formal environment in which we can reason about executions in a sandbox, and we present this now.

The model we present is an extension of work from Dinaburg et al. [52]. We consider execution at the machine instruction level, and since an instruction can access memory and CPU registers directly we consider a system state as the combination of memory contents and CPU registers. Let M be the set of all memory states and C be the set of all possible CPU register states. We denote all possible instructions as I, where each instruction can be considered a machine recognisable combination of opcode and operands stored at a particular place in memory.

A program P is modelled as a tuple (M_P, ϵ_P) where M_P is the memory associated with the program and ϵ_P is an instruction in M_P which defines the entry point of the program. There are often many programs executing on a system and each of these may communicate with each other through the underlying OS. As such, we model the execution environment E as the underlying OS and the other programs running on the system.

We define a transition function $\delta_E : I \times M \times C \rightarrow I \times M \times C$ to represent the execution of an instruction in the environment E. It defines how execution of an instruction updates the execution state and determines the next instruction to be executed. The trace of instructions obtained by executing program P in execution environment E is then defined to be the ordered set $T(P, E) = (i_0, \ldots, i_l)$ where $i_0 = \epsilon_P$ and $\delta_E(i_k, M_k, C_k) = (i_{k+1}, M_{k+1}, C_{k+1})$ for $0 \leqslant k < l$. We note here that the execution trace does not explicitly capture which instructions are part of the program, with the exception of i_0, but rather all the instructions executed on the system including instructions in other processes and the kernel. For any two elements in the execution trace $i_j \in T(P, E)$ and $i_k \in T(P, E)$ we write $i_j < i_k$ if $j < k$, $i_j > i_k$ if $j > k$ and otherwise $i_j = i_k$. We use this to define ordering between the instructions of the sequence.

4.3.1 Malware execution trace

We now introduce the concept of malware execution trace. The goal of the malware execution trace is to define the set of instructions in a sandbox that belongs to the malware under analysis. The malware execution trace must, therefore, determine for each instruction in a given system whether it belongs to the malware or if it is benign.

Suppose P is a malware program and P_A is some malware tracer that aims to collect P's execution trace. Malware program P is interested in evading analysis and gain privilege escalation by using code-reuse attacks and code injections. As such, the execution trace of the malware may contain instructions that are not members of program P's memory M_P.

To monitor the malware across the environment, the malware monitor P_A maintains a shadow memory that allows it to label the memory and the CPU registers. This shadow memory is updated for each instruction in the execution trace. Let $S \subseteq M \times C$ be the set of all possible shadow memories. We then define the propagation function $\delta_A : S \times I \rightarrow S$ to be the function that updates the shadow memory when an instruction executes. The list of shadow memories collected by the malware tracer

is now defined as the ordered set: $ST_A(T(P,E)) = (s_0, \ldots, s_l)$ where $\delta_A(s_k, i_k) = s_{k+1}$ for $0 \leqslant k < l$.

The job of the malware tracer is to determine for each instruction in the execution trace whether the instruction belongs to the malware or not. To do this, the analyser uses the predicate $\Lambda_A : S \times I \rightarrow \{true, false\}$. The malware execution trace is now given as the sequence of instructions for which Λ_A is true and we call Λ_A the inclusion predicate. We define the malware execution trace formally as follows:

Definition 1. *Let $T(P,E)$ be an execution trace and P_A a malware tracer. The malware execution trace is the ordered set $\Pi_A = (m_0, \ldots, m_d)$ where:*

- *Π_A is a subsequence of $T(P,E)$;*
- *$\exists v \mid m_j = i_v \wedge \Lambda_A(s_v, i_v)$ for $0 \leqslant j \leqslant d$.*

The above definition says that the malware execution trace is a subsequence (ordering is preserved) of the entire whole-system trace and for each instruction in the malware execution trace there is a corresponding instruction in the whole-system trace for which the inclusion predicate is true.

The malware execution trace gives us a definition we can use to reason about the properties of malware tracers. In particular, for a given malware tracer it highlights the propagation function, δ_A, and the inclusion predicate, Λ_A, to be the defining parts. Having constructed our model of malware tracers and identified the key aspects that determine how they collect the execution trace, we now move on to present how Minerva precisely captures system-wide propagation.

4.4 Tracing the malware execution

The goal is to capture malware execution in the context of code injections and code-reuse attacks. The overall idea is to use dynamic taint analysis to mark the malware under analysis as tainted and then capture its system-wide execution by following how the taint propagates through the system.

Algorithm 1 gives an overview of our approach to capturing the malware execution trace. Assuming the first instruction executed on the system is the entry point of the malware, the first step **(line 2)** is to taint the memory making up the malware. Next, execution continues until there is no more taint or a user-defined timeout occurs, and for each instruction executed we check if the memory making up the instruction is tainted **(line 8)**. We include the instruction in the malware execution trace if the

instruction is tainted **(line 9)**. In the case the instruction is not tainted **(line 10)** then we check if the instruction is part of a buffer that holds the code-reuse instructions we need to monitor (**line 12**). If the instruction is part of a code-reuse attack, then we set a temporal variable indicating that the instruction should be included in the malware execution trace, and also remove it from the buffer that monitors code-reuse attacks. Next **(line 18)**, we check if the instruction initiates any code-reuse. If it does, then we set the temporal variable indicating the instruction must be appended to the malware execution trace and also include the reused code into our code-reuse buffer. Finally, we emulate the instruction propagate taint accordingly **(line 28)**.

In the following three sections, we describe in detail how we taint the malware initially (Section 4.4.1), how we identify code-reuse attacks (Section 4.4.2), and how we propagate taint (Section 4.4.3).

ALGORITHM 1: Main algorithm

```
   Data: Malware execution trace Π, gadgets 𝒢.
   Result: (input) Malware sample B
 1 // Initialisation
 2 𝒫 ← init_taint(B)
 3 𝒯𝒢 ← ∅
 4 // Full system instrumentation
 5 i ← first_instr()
 6 while 𝒫 ≠ ∅ do
 7 |   // is the instruction tainted?
 8 |   if i[A] ∈ 𝒫 then
 9 |   |   Π ← Π^⟨i⟩
10 |   else
11 |   |   // code-reuse handling
12 |   |   // is it in the gadget buffer?
13 |   |   if i ∈ 𝒯𝒢 then
14 |   |   |   append = true
15 |   |   |   𝒯𝒢 ← 𝒯𝒢\{i}
16 |   |   end
17 |   |   // does it initiate code-reuse?
18 |   |   if initiates_code_reuse(i, 𝒫) then
19 |   |   |   append = true
20 |   |   |   𝒯𝒢 ← get_code_reused(i, 𝒯𝒢)
21 |   |   |   𝒢 ← 𝒯𝒢 ∪ 𝒢
22 |   |   end
23 |   |   if append = true then
24 |   |   |   Π ← Π^⟨i⟩
25 |   |   end
26 |   |   append = false
27 |   end
28 |   (i, 𝒫) ← update(i, 𝒫)
29 end
30 return (Π, 𝒢)
```

4.4.1 Initial setting

We consider malicious code execution on the basis of tainted memory. The only two ways we include instructions in the malware execution trace is if the memory that makes up an instruction is tainted or if the instruction is part of a code-reuse attack. Furthermore, apart from the initial taint, the only way we introduce new taint in the system is from instructions that are already tainted. The initial taint is, therefore, the seed for the rest of the analysis and has a significant impact on the completeness and precision of the analysis.

The goal is to ensure completeness and the initial taint must, therefore, cover all memory that belongs to the malware code base, and also the memory from where the malware can derive dynamically generated code. If we miss any such memory, then there is a possibility the malware uses this memory to generate code dynamically, and our monitor will miss out when this code executes. On the basis that malware can contain code anywhere in its module, we taint the entire module of the malware, including data-only sections. The tainting occurs when the program is loaded into memory to ensure our tainting happens before any of the malicious code executes.

4.4.2 Code-reuse identification

The similarity between the code-reuse techniques introduced in Section 4.2.3 is that a benign indirect branch instruction directs execution to another benign instruction but the adversary controls the value that determines the destination. The difference between the techniques is the instruction used i.e. whether it is a `ret`, `jmp` or `call` instruction. Because of this similarity, we consider code reuse attacks on a more abstract level. Formally, we define code reuse attacks as pairs $(\mathcal{GI}, \mathcal{GC})$ where $\mathcal{GC}$ is the reused code *(gadget code)* and $\mathcal{GI}$ is the instruction that initiates the branch to the reused code *(gadget initiator)*. When malware reuses code as part of their control-flow, $\mathcal{GI}$ is the trigger that allows the malware to use $\mathcal{GC}$ for its purposes.

We identify $\mathcal{GI}$ instructions as instructions made up of non-tainted memory that branches to non-tainted memory, but the memory that determines the destination of the branch is tainted. The particular execution pattern we capture with this definition is malware that creates a control-flow by chaining two benign code regions (non-tainted code) together by overwriting the address determining the branch destinations and not the code itself. Table 4.1 illustrates the specific rules we use to determine if an instruction is a $\mathcal{GI}$.

Address	**Instruction**	**Code-reuse conditions**	GI
I	`CALL [A]`	$A \in \mathcal{P}, [A] \notin \mathcal{P}, I \notin \mathcal{P}$	✓
I	`CALL reg`	$reg \in \mathcal{P}, [reg] \notin \mathcal{P}, I \notin \mathcal{P}$	✓
I	`JMP [A]`	$A \in \mathcal{P}, [A] \notin \mathcal{P}, I \notin \mathcal{P}$	✓
I	`JMP reg`	$reg \in \mathcal{P}, [reg] \notin \mathcal{P}, I \notin \mathcal{P}$	✓
I	`RET`	$esp \in \mathcal{P}, [esp] \notin \mathcal{P}, I \notin \mathcal{P}$	✓

Table 4.1: Conditions for identifying $\mathcal{GI}$ instructions. $\mathcal{P}$ is the set of tainted memory.

When we identify a $\mathcal{GI}$ instruction, we proceed to determine $\mathcal{GC}$. Previous literature on exploit mitigation has tested several ways on how to define $\mathcal{GC}$, and there is no definitive best solution. The difficulty occurs because $\mathcal{GC}$ in practice can include an arbitrary number of instructions, have arbitrary structure, and can vary between small gadgets and entire functions. To this end, we monitor if the destination of $\mathcal{GI}$ is a function, and if so, consider the $\mathcal{GI}$ to be a function call inside the malware execution trace, and if the destination is not a function, then we include $\mathcal{GI}$ in the execution trace as well as the instructions of the first basic block at $\mathcal{GI}$'s destination.

For each $(\mathcal{GI}, \mathcal{GC})$ pair that we capture, we must include the instructions of the pair in the malware execution trace. However, the gadget is only a one-time execution so we cannot taint the instructions as we do with regular malware code. Instead, we append the $\mathcal{GI}$ instruction to the malware execution trace as we observe it, and put all the instructions of the $\mathcal{GC}$ basic block into a buffer. Instructions in the buffer are then included in the execution trace only the first time they are executed right after the $\mathcal{GI}$ instruction execution. In this way, we only include gadgets at the specific moment the malware uses them.

We notice here that there is one exception to the rule, which is when we observe a chain of code-reuse attacks. We observe this exception in code injection techniques where a chain of code-reuse attacks is used by the malware to transfer execution to shellcode. To identify this chain, we monitor for chained code-reuse attacks, and if the last instruction of the chain transfer execution to tainted memory, then we include this in the chain.

4.4.3 Propagation function

The core part of our propagation function is defined by our *update* algorithm, shown in Algorithm 2. The first step is to apply the taint propagation for a given instruction. Next, we continue to execute (emulate) the instruction, and if the instruction is part of tainted memory, then we also taint the output of the instruction. We do

this because some malware generates dynamically generated code by reading benign code and modifying it for malicious purposes. In fact, our running example of the Gapz malware employs this type of behaviour. We leave this Section to the taint propagation aspects specific to the algorithms in Minerva, and then in Section 4.6 we describe in detail the specific policies of the taint implementation that we use in Minerva.

ALGORITHM 2: update

Data: Instruction i, memory propagation set P.
Result: Propagation set $\mathcal{P}$

```
// Initialisation
P ← propagate_taint(i, P)
i_next ← exec_instr(i)
if i[A] ∈ P then
    for o ∈ i[O] do
        P ← P ∪ {o}
    end
end
return (i_next, P)
```

4.5 Identifying code injections

In this section, we present our approach to identifying code injections in the malware execution trace and also extract intrinsic characteristics about these by performing semantics-aware dependency analysis. The approach consists of three steps. First, identifying control-flow transitions from one process to another and arrange such findings into transition pairs. Second, finding any code-reuse attacks involved in the transition and, third, performing backward dependency analysis on these two components to generate a code injection graph. We detail each of these steps below.

4.5.1 Transition model

The transition model captures key instructions within the malware execution trace that enable malware execution to flow from one process to another. This type of flow is composed of an initiator instruction in the source process and a target instruction in the destination process. Together, the initiator and target instruction make up a transition pair. Identifying the transition pairs is not trivial because the instructions in the malware execution trace are ordered by when they were observed in a single-core execution environment and not necessarily the control flow of the malware

execution. In practice, this problem of misalignment between control-flow and observation time occurs because of asynchronous procedure calls, parallel execution and context switches.

Previous works have handled the problem of identifying transition pairs with two different solutions. Ugarte et al. [147] create a transition pair whenever a thread context switch occurs. We have found that in practice this approach is inaccurate because it generates too many transition pairs. The reason is that context-switches do not reflect control-flow transitions from the process or thread being switched out to the one that switches in. Instead, context-switching is merely a technique to ensure scheduling amongst processes and threads on the system. Bonfante et al. [21] hook a set of function calls which are known to initiate execution in remote processes. Although this approach works well in practice for known injection techniques, it lacks generality because it cannot identify unknown code injections.

To overcome the limitations of previous work, we first identify all target instructions in a general manner independent of function hooking and then proceed to identify initiator instructions. In comparison to previous work that first identifies the initiator instruction and then the target instruction, we can identify the target instruction independently of the initiator instruction because we rely on taint to trace the malware execution. We identify a target instruction as the first malware instruction executed in each process that is a non-code-reuse attack. For such instructions, we know injection has occurred because this is the first time the malware executes in the process.

For each target instruction, we trace backwards in the malware execution trace to determine the respective initiator instruction. We use function hooking to identify if the injection matches an already known injection technique. If it does, then we mark this as the code injection, and if it does not, then we deploy a general heuristic to identify the initiator instruction. In particular, during execution, we monitor all API calls done by instructions in the malware execution trace, including obfuscated calls such as `push X; rol[esp], Y; ret`. We keep a subset of these calls in a set F with all the calls to functions that we know possibly initiate execution in a remote context e.g. `CreateRemoteThread`, `ResumeThread`, `CreateProcess`, `QueuseUserAPC`. For each target instruction, we check if there is an element $e \in F$ that initiates execution of the given target instruction. If there is, then we declare the element e as the initiator of the code injection. However, if there is no such element, then we have a code injection that relies on an unknown method for injecting the code. In this case, we trace back in the malware execution trace to the first instruction that

branches to something outside of the malware memory (such as calling a function in a dynamically loaded module) and is a non-gadget instruction. We then declare this as initiator instruction.

It is important to note here that the hooking is only used to identify the injection initiator and not to detect that a code injection has happened. Our approach to determining whether a code injection has occurred is entirely independent of the hooking.

4.5.2 Injection mechanics

After finding a pair of initiator and the target instructions, we continue to identify if there any code-reuse occurred between them in the execution that are of importance to the injection. Specifically, we consider each code injection as a sequence of instructions $\langle initiator, R, G, target \rangle$ where R is a sequence of instructions with process identifier (PID) not equal to the PID of $target$ and G is a sequence of instructions that are all code-reuse pairs and have PID to be the same as the PID of $target$. Both R and G may be empty sequences. We call G the injection catalyst and the tuple $(initiator, catalyst, target)$ the key components of the code injection. The key components of the code injection make up the control flow of the code injection: $initiator \rightarrow catalyst \rightarrow target$.

Figure 4.3 shows the key components of the code injection collected by Minerva when matched with the Gapz malware sample. The initiator instruction here is the instruction `call SendNotifyMessage` at address `9b3b00` in the malware process. This call triggers three indirect `call` gadgets that transfer execution to 6 ROP gadgets. The target instruction is `mov ebp, esp` at address `77ef48c0`.

In addition to the injection mechanics, there have recently been proposed execution purely via code injection attacks [149]. To support identifying this type of execution, Minerva also comes with an option for capturing code injections in case we observe a code-reuse chain above a given length. However, this feature is more esoteric and we have not found any malware that purely relies on code-reuse attacks.

4.5.3 Code injection graph

The key components give valuable insight into a code injection and the instructions that are part of it. However, on their own, the key components provide little insight into how the malware established these components. To give insights about this, we

PID	**Address**	**Instruction**
4d0	9b3b00	`call SendNotifyMessage`
5f0	1001b4b	call [eax] ; ***KiUserDispatcher***
5f0	1001b59	call [eax + 8]
5f0	1022599	std
5f0	102259a	ret
5f0	1001b6e	call [eax + 4]
5f0	7c9ee5be	mov ecx, 0x94
5f0	7c9ee5c3	rep movsd
5f0	7c9ee5c5	pop edi
5f0	7c9ee5c6	xor eax, eax
5f0	7c9ee5c8	pop esi
5f0	7c9ee5c9	pop edi
5f0	7c9ee5ca	ret
5f0	77ec5b26	cld
5f0	77ec5b27	ret
5f0	101179c	pop eax
5f0	101179d	ret
5f0	7c9015f8	`__alloca_probe`
5f0	7c90160c	ret
5f0	7c802213	`WriteProcessMemory`
5f0	7c802298	ret
5f0	101179c	pop eax
5f0	101179d	ret
5f0	1002080	jmp eax
5f0	77ef48c0	mov ebp, esp

Figure 4.3: Key components of Gapz code injection.

```
lea edi, [esp + 0x10]
pop eax
call eax
```

Figure 4.4: Code of KiUserDispatcher.

construct the code injection graph. The code injection graph describes the control-flow of the code injection and the data-flow of tainted memory involved in the code injection. The nodes of the graph are either instructions in the malware execution trace or non-malicious instructions that propagate taint. The edges in the graph, therefore, show control-flow or taint-flow. In practice, we also annotate the nodes with several meta-descriptions such as their given process, module and function.

To construct the code injection graph, we analyse the taint-propagation history of the tainted memory in the catalyst and the target from the key components of a given injection. Specifically, we trace backwards in the propagation history of the tainted memory involved in the catalyst and the target until the instruction that propagates the tainted memory is part of the malware execution trace or the propagated memory is part of the original memory when the malware was first loaded. The result is a graph showing how the malware established the memory involved in the key components, giving complete insight into the control-flow and data-flow of the given code injection.

4.6 Taint implementation

Dynamic taint analysis is at the core of our technique, and we now clarify central elements of the taint implementation in Minerva, which is a slightly modified version of the taint analysis deployed by PANDA. From a high-level point of view, PANDA raises the QEMU intermediate representation into LLVM code and then performs taint analysis on the LLVM code itself using a byte-level granularity.

PANDA has extensive support for disk tainting. Specifically, PANDA has several hooks in QEMU's implementation of the Integrated Device Electronics (IDE) interface that PANDA uses for its record-and-replay facility. Based on these hooks, PANDA provides several callbacks that plugins can use to get notified whenever disk-related events occur, and these are the specific callbacks that the taint system uses to support tainting across various storage components. The taint system maintains a shadow memory that itself has several sub-shadow memories representing various components, including hard disk, I/O buffer, RAM and registers. These partitions are then maintained by way of the callbacks mentioned above, including during data transfers from one storage unit to another, e.g. from disk to RAM. We refer to [53] and [153] for more detailed presentations of PANDA's taint system.

By default, Minerva does not use pointer tainting, and this is also the case for all empirical work involving Minerva that we present in this thesis. Furthermore, Minerva makes no effort to identify implicit data flows exhibited by the malware.

In this way, Minerva takes a conservative approach to false positives and aims for minimal risk of over-tainting. However, it also leaves Minerva vulnerable to more false negatives since malware can exploit our conservative policies, which we discuss further in Section 4.8.

The only change we made to PANDA's taint implementation is decreasing the size of selected data structures in the shadow memory. Specifically, PANDA supports taint labelling at the cost of large data structures holding information about each tainted byte. However, we do not need taint labels and, therefore, stripped these aspects out.

4.7 Evaluation

To verify the effectiveness of our approach, we evaluate it against a set of benchmark applications comprising synthetic applications and real-world malware. In Section 4.7.3, we empirically validate the correctness of our approach by matching it with applications where we have ground-truth, and in Section 4.7.4 we compare our approach with previous work. Our results show we find more genuine code injections and that the taint does not explode. In Section 4.7.5 we perform in-depth analysis on two case studies, demonstrating that our approach can capture malware propagation with code injections and code-reuse attacks, and also give valuable detailed insights about the propagation. A checklist for prudent evaluation is given in Table B.1 in Appendix B.

4.7.1 Data collection

In total, we evaluate Minerva against three data sets comprising 46 applications. These consist of applications implemented by ourselves, real-world malware and also public benchmarks.

1. **Benchmark #A : Ground truth code injection techniques.** The first set, A, is composed of our implementations of several publicly-document code injection techniques. In this set four applications use code-reuse attacks, and six do not. Table 4.2 lists the specific Windows API functions used by the six applications without code-reuse attacks use.

2. **Benchmark #B : Analysed real-world malware.** The second set, B, is composed of a set of malware samples from four different malware families where anti-malware companies have documented the code injections of the samples.

We ensure correct samples by only selecting those with hash sums reported in the specific malware analysis reports.

3. **Benchmark #C : Benign single process applications.** The third set, C, composes benchmark applications we are sure do not inject code into other processes. All of the samples in data set C are from the WCET benchmark suite [70].

4.7.2 Experimental set up

We conduct all of our Minerva experiments on a 4-core Intel-7 CPU with 4.2 GHz and a Windows 7, 32-bit guest architecture. The guest is in a closed network and connected to another virtual machine that performs network simulation using INetSim[76]. As such, malware samples that connect back to some CC server will be able to resolve DNS names, connect to every IP and also receive content. However, the content itself is the default data provided by INetsim.

We executed the applications on the guest machine with a local admin account, and User Account Control (UAC) enabled. We perform no user stimulation during the analysis, and there were no applications apart from the generic Windows processes running in the guest machine itself.

4.7.3 Experimental validation of correctness

In our first experiment, we empirically evaluate the correctness of Minerva. We use Minerva to analyse the samples in our data sets and match the results with ground truth. The first five columns of Table 4.3 show our results of matching Minerva with the samples from set A and B. For the malware samples in data set B we manually verify each process propagation identified by Minerva.

4.7.3.1 True positives

Minerva correctly identifies when code injection occurs in all of the code injecting binaries and also identifies when code-reuse attacks are part of the code injection techniques. As such, the true positive rate is perfect, and we do not miss any system-wide malware execution.

ID	Windows API functions used.
A1	`OpenProcess, WriteProcessMemory, CreateRemoteThread`
A2	`OpenProcess, WriteProcessMemory, SetThreadContext, ResumeThread`
A3	`OpenProcess, WriteProcessMemory, QueueUserAPC`
A4	`OpenProcess, MapViewOfSection, SetThreadContext, ResumeThread`
A5	`OpenProcess, MapViewOfSection, CreateRemoteThread`
A6	`OpenProcess, MapViewOfSection, QueueUserAPC`

Table 4.2: Details about the Windows API functions several of the applications of data set **A** use to perform code injection.

			Minerva		CO[21]	CS[51]
Samples	# num	CRI	CRI	CI	CI	CI
(A) PowerLoader	1	✓	1	1	0	0
(A) PowerLoaderEx	1	✓	1	1	0	0
(A) AtomBombing	1	✓	1	1	0	0
(A) Codeless	1	✓	1	1	0	0
(A) A1	1		0	1	1	1
(A) A2	1		0	1	0	1
(A) A3	1		0	1	0	0
(A) A4	1		0	1	0	1
(A) A5	1		0	1	1	1
(A) A6	1		0	1	0	0
(B) CryptoWall 4 [3]	4		0	4	0	4
(B) Gapz [130]	4	✓	3	4	1	1
(B) Ramnit [129]	8		0	8	3	5
(B) Tinba [98]	9		0	9	8	8
Total	**35**		**7**	**35**	**14**	**22**

Table 4.3: Evaluation with code injecting binaries.

4.7.3.2 False positives

Minerva relies on taint propagation to capture malware execution, and it is imperative that the taint does not explode, resulting in a high number of false positives. Minerva finds no code injecting binaries when matched with our data set C, as shown in Table 4.4. These samples are, however, reasonably trivial and do not engage in any complex system-interactive behaviours. In order to measure the number of false positives Minerva produces against more complex applications we match for each malware sample in data set B the processes with malware execution as reported by Minerva to those from the reports produced by the anti-malware companies. In addition to this, we manually verify if each process propagation reported by Minerva is a true or false positive. Table 4.5 lists the results and, in total, we monitored 77 processes across the samples, of which 52 processes are due to multi-process propagation. We found four false-positive processes. These processes occur in three samples, and two of the processes only execute one and eleven "malicious" instructions, respectively.

We consider these results to show that Minerva captures the malware execution with a high level of precision. Although we find several false positives, we do not think these are severe, but rather represent a modest limitation for the following reasons. First, we find 4 out of 77 processes to be false positives; however, two of these processes only execute 1 and 11 malware instructions and are easy to discard in a follow-up analysis, leaving us with two false positive processes in one sample. On a more detailed level, we observe a total of 236207 unique tainted malicious instructions amongst all samples in our data set and 1480 of these occur within processes that are false positives. As such, it is only 0.006% of the instructions that are from false positive processes, which we consider to be a low number. Second, the tracing capabilities presented in this chapter are *raw* in that we do not deploy any analysis or heuristics on the traces to identify potential false positives. There are many options available for such heuristics and in Section 6.3.5 we come up with a solution that, in important cases, verifies if code execution reported as malicious is similar to code that was in the system prior to malware execution and if so (with a few more conditions), considers it to be a false positive and discards it. This solution eliminates all the false positives we observe in this chapter.

In the following paragraphs, we give detailed accounts for our manual analysis of the false positives observed in data set B. In [98] CSIS and Trend Micro reports that the Tinba malware injects into `winver.exe`, `explorer.exe`, `svchost.exe`,

		Minerva	CO[21]	CS[51]
samples	# num	CI	CI	CI
(C) WCET [70]	11	0	0	0

Table 4.4: Evaluation with non code-injecting binaries.

`firefox.exe`, `iexplore.exe` and `firefox.exe`. In eight out of nine analysed samples, Minerva found injections into `winver.exe` and `explorer.exe`, and in one sample, Minerva found injections into seven processes on the system. This sample injects all processes on the system for which it has privileges. We observed one process that crashes, and this resulted in over-tainting occurring inside of the Windows process `WerFault.exe`. However, Minerva only observed one instruction inside this process. The reason we did not see any injection into any of the browsers was that the system had no such processes running. However, we ran the same experiment but with three processes executing on the system with names `firefox.exe, iexplore.exe, chrome.exe`, respectively, and found eight out of the nine samples inject into at least one of these browser processes.

In [130] researchers from Eset report that Gapz injects code into the Windows process `explorer.exe` and also uses code-reuse attacks. Minerva found code injections into `explorer.exe` in three of the four analysed samples. These three samples all used code-reuse attacks in their injection. In two of these samples, Minerva captured `explorer.exe` to be the only process where injection occurred. In the third sample, Minerva also found execution inside of `sysprep.exe` and from manual reverse engineering of the sample, we discovered that this is because the sample tries to bypass Windows User Account Control (UAC) by executing code inside of the trusted process `sysprep.exe`. The last sample did not use code-reuse attacks, and injected code into `svchost.exe` rather than `explorer.exe`. We verified manually that this malware does indeed inject code into `svchost.exe` and relies on `CreateProcessInternalW` and `ResumeThread` functions, and no code-reuse attacks, to perform the injection.

In [129] researchers from Symantec report that Ramnit injects code into `svchost.exe` or `IEXPLORE.EXE`. In seven out of the eight Ramnit samples, Minerva found injections into `svchost.exe`. In one sample Minerva also captured injection into `samplemgr.exe` and one sample crashed resulting in eleven instructions executing inside of `WerFault.exe` and we verified this is indeed a false positive. Because we did not have any `IEXPLORE.EXE` processes running on the system, we analysed several of the Ramnit samples manually to investigate if they were supposed to inject into `IEXPLORE.EXE`. We found that, similar to the findings reported by Symantec, that the

samples would first try to inject into `svchost.exe` and only if this was unsuccessful they would inject into `IEXPLORE.EXE`.

In [3] researchers from Cisco report that CryptoWall samples inject into `svchost.exe` and `explorer.exe`. In three samples, we observed code injection into these two processes, and in one sample, we observed the malware injecting code into `credwiz.exe`. One of the samples had code execution in `vssadmin.exe` and `svchost.exe`, and we found these to be false positives. We found the `vssadmin.exe` to be a false positive because the instructions executed in `vssadmin.exe` that Minerva declared malicious corresponded to the real instructions of `vssadmin.exe` itself.

4.7.4 Comparative evaluation

In this section, we present our second experiment. The goal of this experiment is to assess the quality of our results in comparison to existing work. Specifically, we compare Minerva to state-of-the-art analysis tools in the form of a recent fine-grained malware disassembler, Codisasm [21], and a coarse-grained malware analysis platform, Cuckoo Sandbox.

Codisasm is a disassembler that relies on both static and dynamic analysis and aims to disassemble binaries with self-modifying code and overlapping instructions. The tool is built on top of PIN and hooks two functions from the Windows API `CreateRemoteThread` and `CreateRemoteThreadEx` to follow the malware propagation. Although Codisasm is a malware disassembler, it relies on dynamic analysis to capture the instructions executed by the malware in order to perform its disassembly. Cuckoo is a Sandbox that analyses malware from user-mode at the API call granularity and contains many techniques for automatically tracing malware in case of code injections. As such, both Codisasm and Cuckoo Sandbox follow malware based on function hooking and heuristics about known API calls.

4.7.4.1 Comparative evaluation of correctness

The results in terms of capturing code injections are shown in Table 4.3. Minerva outperforms both Cuckoo Sandbox and Codisasm by a large margin. Cuckoo Sandbox detects code injection in 22 samples and Codisasm in 14 samples out of 35 applications in total where Minerva detects code injection in all of the samples.

In comparison to Cuckoo Sandbox, the first thing we notice is that Cuckoo Sandbox fails on all four synthetic injection techniques that rely on code-reuse attacks. Furthermore, Cuckoo fails to observe code injection in three of the Gapz samples

Family	Sample	#Processes	#Instrs	False positive
Gapz	33d154d84e830aa18973b04e64879466	sample.exe svchost.exe	2535 2183	
Gapz	e5b9295e0b147501f47e2fcba93deb6c	sample.exe explorer.exe sysprep.exe	6564 4673 1298	
Gapz	0ed4a5e1b9b3e374f1f343250f527167	sample.exe explorer.exe	6308 1756	
Gapz	089c5446291c9145ad8ac6c1cdfe4928	sample.exe explorer.exe	5904 3671	
TinyBanker	b062be1e561c20b6fb829ad9a3303431	sample.exe winver.exe explorer.exe	229 583 731	
TinyBanker	b6991e7497a31fada9877907c63a5888	sample.exe winver.exe explorer.exe dwm.exe taskhost.exe cmd.exe conhost.exe dllhost.exe	406 1074 771 519 519 519 519 519	
TinyBanker	0e252ec52d7f4604d6b8894e479de233	sample.exe winver.exe explorer.exe	248 583 729	
TinyBanker	d1c13acddb7c13d0cf5a5c49e53a2906	sample.exe winver.exe explorer.exe	303 583 731	
TinyBanker	08ab7f68c6b3a4a2a745cc244d41d213	sample.exe winver.exe explorer.exe	300 583 731	
TinyBanker	8e8cd6dc7759f4b74ec0bfa84db5b1a5	sample.exe WerFault.exe	1263 1	 ×
TinyBanker	c141be7ef8a49c2e8bda5e4a856386ac	sample.exe winver.exe explorer.exe	300 229 729	
TinyBanker	debfdbd33d6e4695877d0a789212c013	sample.exe winver.exe explorer.exe	300 229 729	
TinyBanker	6244604b4fe75b652c05a217ac90eeac	sample.exe winver.exe explorer.exe	301 583 731	
CryptoWall	e28a0ed74e78e75710b0d46742e407e3	sample.exe explorer.exe svchost.exe	11085 6548 8224	
CryptoWall	48e4daf494e4fa2577d8fa94b7b89e35	sample.exe credwiz.exe	10847 233	
CryptoWall	e73806e3f41f61e7c7a364625cd58f65	sample.exe explorer.exe svchost.exe vssadmin.exe svchost.exe	5413 6964 11462 1125 343	 × ×
CryptoWall	5384f752e3a2b59fad9d0f143ce0215a	sample.exe explorer.exe svchost.exe	5425 6548 17210	
Ramnit	16f678a9f654d396d0adc8c7011f272e	sample.exe samplemgr.exe svchost.exe svchost.exe	1266 6278 1748 5049	
Ramnit	448ce1c565c4378b310fa25b4ae3b17f	sample.exe svchost.exe svchost.exe	6316 1748 5049	
Ramnit	971cb5f32a2c09ab6e69eef612801ab4	sample.exe svchost.exe svchost.exe	6519 1748 5049	
Ramnit	9380683d2c2903848514a7ca884cfc0f	sample.exe svchost.exe svchost.exe	6416 1748 4673	
Ramnit	2fd2dcba0b787961f6497e5c106a167c	sample.exe svchost.exe svchost.exe	6447 1748 4673	
Ramnit	8f0eb8299491d218e346379b7964ff50	sample.exe svchost.exe svchost.exe	6362 1748 4673	
Ramnit	8b73198992c98f26581a3d81ab1e8e94	sample.exe WerFault.exe	236 11	 ×
Ramnit	3bb86e6920614ed9ac5d8fbf480eb437	sample.exe svchost.exe svchost.exe	8220 1748 5098	

Table 4.5: The results of running data set **B** through Minerva. The columns show the family, md5 sum of sample, the processes in which execution occur, the number of tainted malicious instructions and whether the process is a false positive.

and three of the Ramnit samples. The sample from the Gapz family that Cuckoo correctly identifies to have code injection is the sample without code-reuse attacks as described above. Both Minerva and Cuckoo Sandbox accurately reported the use of `ResumeThread`.

The second thing we notice when comparing Minerva to Cuckoo is that Cuckoo failed to correctly identify code injections that use remote procedure calls (RPC) via QueueUserAPC. We believe that this is because many RPCs do not constitute code injections, so labelling each of them as such will produce many false positives. Minerva does not run into this problem because tainted code must be executed for Minerva to declare that a code injection occurs and can, therefore, identify which RPC result in a code injection. Furthermore, it may seem that Minerva should identify a code reuse attack for the RPC because an indirect branch in the target process transfers execution to a value set by the process sending the RPC. However, the destination of the indirect branch is tainted code, so the conditions for a code-reuse attack, as described in Section 4.4.2, are not satisfied. It is important to note here that even in cases where malware initiates an RPC to non-tainted memory, Minerva will not declare it as a code injection because only one code-reuse attack will be observed. As such, the conditions for code injection, as described in Section 4.5, are not satisfied.

Codisasm finds code injections in 14 of the 35 samples. In particular, Codisasm finds code injections in both of the synthetic samples that use `CreateRemoteThread`. Because we do not have direct access to their system, but rather through a web interface[2], it is difficult to assess the specific cause of limitations in the other samples. We would often get error messages back from the server that either an unknown malfunction occurred or the timeout occurred because the system had crashed or was not able to produce any traces. However, we consider the limitations to be a result of two properties: one conceptual and one in the implementation. In terms of conceptual limitation, Codisasm follows malware propagation based on monitoring for calls to `CreateRemoteThread` and `CreateRemoteThreadEx`. However, many techniques do not use either of these API functions to inject code - notably, only two out of ten injection techniques in benchmark A use `CreateRemoteThread`. From an implementation point of view, the dynamic analysis component of Codisasm relies on PIN. PIN is a process-level dynamic binary instrumentation framework and has previously been reported to be unreliable when instrumenting malware and also fails instrumentation in case of injection into processes with multiple active threads [68].

[2]http://codisasm.lhs.loria.fr/

4.7.4.2 Comparative evaluation of performance

To give a fair and meaningful comparison, we must also consider the performance of each of these tools. The web interface exposing Codisasm sets an upper limit of two minutes analysis time, and this is what we used. In our experiments using CuckooSandbox, each analysis took about three minutes. For Minerva, we recorded each execution of the samples in data set B for 25 seconds and had an average replay time of 3197 seconds and an average of 1614 seconds to replay 95% of the malicious instructions.

It is clear that the analysis behind Minerva is substantial and we do not claim to outperform, or even come close to, systems like CuckooSandox that deploys a coarse granular approach or even Codisasm that relies on DBI. Throughout the development of our ideas and techniques, we have focused on a fine level of granularity and have paid less attention to performance. This also highlights where Minerva is useful: in scenarios where more efficient dynamic analysis approaches fail and as assistance to manual analysis. Minerva can certainly be used in automated procedures and frameworks, but we do not foresee it analysing thousands of samples each day. Minerva focuses on a fine-grained analysis and it pays the price on performance. We give a comprehensive performance evaluation of Minerva in Section 5.6.7 and discussion on the limitations and possible improvements hereof in Section 5.7.

4.7.5 Detailed analysis

In this section, we present two detailed case studies of Minerva to demonstrate the precision of Minerva and its ability to capture intrinsic characteristics about code injections. The first case is from a Gapz malware sample, and the second case is from a recently published code injection technique called AtomBombing [106].

Gapz malware. Minerva identifies a single code injection when analysing a sample from the Gapz malware family. The sample injects code from its original process into `explorer.exe` and we know there are code-reuse attacks involved in the injection because Minerva recognises the use of an injection catalyst. Figure 4.3 shows the key components identified by Minerva, and Figure 4.5 shows a part of the code injection graph for the three first gadgets in the key components. Boxes with rounded corners display control-flow nodes and squared boxes display taint-propagating nodes. The boxes with rounded corners show the code-reuse initiator and also the gadget itself.

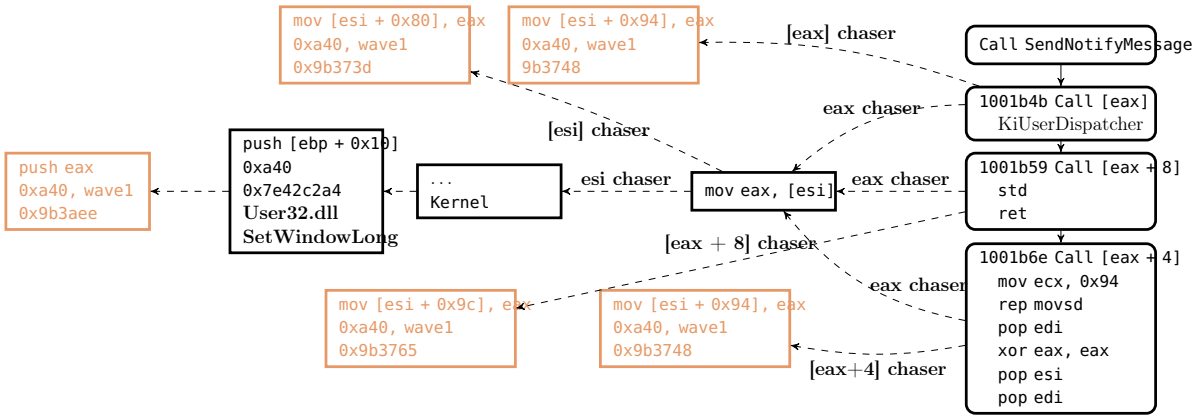


Figure 4.5: Code injection graph for the first three gadgets in Gapz code injection. The graph shows that in all three `call` gadgets the taint in *eax* was propagated through `SetWindowLong` in the host process.

Investigating the key components, we observe the first code-reuse attack is a call to `KiUserDispatcher`, and the bottom of Figure 4.4 shows the code of this function. This gadget puts the value of `esp + 0x10` into `edi`. The second gadget executes the instruction `std`, which will set the direction flag. The third gadget, initiated by the instruction at `1001b4b`, executes the two instructions: `mov ecx, 0x94` followed by `rep movsd`, effectively causing `0x94` bytes to be copied from `esi` to `edi`. Because the direction flag is set, `edi` and `esi` will be decreased by one after every `mov` instruction. Recall that `edi` was assigned `esp + 0x10` by the first gadget, which means that the memory at the top of the stack will be overwritten with whatever `esi` points to. We can therefore easily conclude the stack is being overwritten with the memory pointed to by `esi`, hinting strongly towards a set of indirect call instructions being hijacked to allow for a ROP attack. Before we proceed, it is important to note that the malware execution trace is a subsequence of all the instructions executed. When the third gadget executes, there have been several push instructions between the first and the third gadgets, resulting in a larger distance between `esp` and `edi` than `0x14` as it appears to be from looking at the malware execution trace. However, some of the instructions that modify `esp` are not part of the malware execution trace and, therefore, not shown by Minerva.

Investigating the code injection graph, we observe that Minerva correctly identifies `SendNotifyMessage` as the code execution initiator and `SetWindowLong` as a function responsible for data-flow in the overwritten addresses in the code-reuse attacks. Continuing analysis reveals that the ROP chain uses WriteProcessMemory to overwrite memory inside the `explorer.exe` process and then proceeds to transfer execution

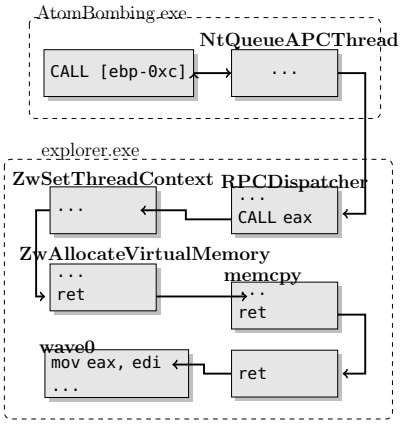


Figure 4.6: AtomBombing injection caught by Minerva with hook on NtQueueAPCThread.

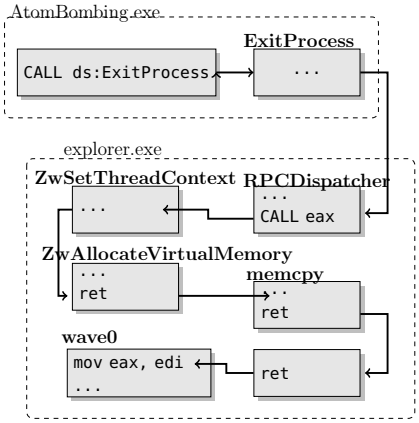


Figure 4.7: AtomBombing injection caught by Minerva without any hooks.

to the malicious code written into `explorer.exe`. Investigating the complete code injection graph further, shown in Appendix A, we see with minimal effort that the code-reuse attacks do turn into a ROP chain and also that `rep movsd` instruction overwrote the return addresses.

AtomBombing. Recently, researchers discovered a new injection technique called AtomBombing [106], which uses code-reuse attacks to avoid standard API calls for code injection. The technique was first presented in October 2016, and only four months later researchers discovered a new 64-bit version of the Dridex malware that had adopted AtomBombing into its arsenal [16].

AtomBombing abuses the global atom table in Windows to share memory between processes and undocumented asynchronous procedure calls to force the injected application to call various functions on behalf of the injecting process. Specifically, the

injecting process writes a ROP chain and shellcode onto the global atom table using `GlobalAddAtom`. The injector then uses `NtQueueApcThread` to force the injected process to call `GlobalGetAtomName` to store the ROP chain and shellcode inside the target process. To invoke execution, the injector again uses `NtQueueApcThread` to force the injected process to call `SetThreadContext` to set `eip` and `esp`. Specifically, the injected process sets `eip` to `ZwAllocateVirtualMemory`, which is the first code-reuse attack in the target process, and `esp` to point to the beginning of the ROP chain. As such, AtomBombing achieves code execution with a combination of calls to `NtQueueApcThread` and `GlobalAddAtom` in the injector process.

Minerva captures the details of AtomBombing and finds the code injection shown in Figure 4.6. Minerva precisely identifies the process transition in AtomBombing, i.e. that a call to `NtQueueApcThread` in the host process causes the target process to call `SetThreadContext`. The `ret` instruction inside `ZwAllocateVirtualMemory` is the first ROP gadget in the injection. This ROP gadget transfers execution to `memcpy` and the `ret` instruction of `memcpy` transfers execution to a simple ROP gadget consisting of only a `ret` instruction. This gadget is there because the ROP chain must catch the `dest` parameter given to `memcpy`, which is 4 bytes away from the original return address. Therefore, the third `ret` instruction executed results in transfer of control to the destination of the copied buffer.

In Figure 4.7, we show the AtomBombing injection caught by Minerva when there are no hooks on `NtQueueApcThread`. In this case, Minerva catches the same code injection except for the initiator instruction, which in this case is a call to `ExitProcess`. The reason this happens is that `NtQueueAPCThread` is an asynchronous procedure call, which means that the last API call in the injector process at the time the injection happens in `explorer.exe` is not `NtQueueAPCThread` because execution has continued inside the injector process itself. This clearly shows the use of hooks in our technique, namely, to capture the correct initiator instruction and not to identify whether an injection has occurred or not.

4.8 Limitations

Evading Minerva. On a fundamental level, our techniques rely on taint analysis for capturing malware propagation. As such, Minerva is limited by the limitations of taint analysis itself, meaning an attacker can deploy information-flow evasive behaviours in order to avoid analysis. This is a common theme for any system relying on taint analysis and has been discussed in-depth by the community [34, 39].

Additionally, from a practical point of view, malware can detect the use of dynamic analysis itself, for example by detecting the presence of QEMU, and indeed this is a problem shared amongst all dynamic malware analysis platforms. However, we do not perform any activities to combat these types of behaviours, and a start would be to harden QEMU by removing QEMU-specific artefacts. A more general approach is to improve the recording stage to rely on a more transparent platform like Ether [52], which is a hypervisor-based malware analysis platform that aims explicitly to hide itself from the malware.

A particular limitation to the record-and-replay paradigm is that we are unable to modify the malware execution, even if we know the malware is trying to evade analysis. Naturally, we can identify the evasion attempt during replay. However, this will not have an immediate effect as we can not "go live". Interesting avenues for future research is to investigate more convenient approaches to incorporate evasion-aware tactics within the recording stage itself or use observations from a detailed analysis via replay as guidance for future recordings.

Target architectures and operating systems. Currently, Minerva supports 32-bit x86 and extending the implementation to 64-bit is something we look to do in the near future. We already have an early version for x64 systems that partially works in that it can follow malware execution, but it is unreliable and crashes early during analysis. The techniques we presented in this Chapter are conceptually applicable to 64-bit as well. However, creating the sandbox environment for 64-bit requires a solid engineering effort. In particular, on Windows 64-bit virtual machine introspection is more demanding than on 32-bit because 64-bit Windows can execute both 32-bit and 64-bit applications. Windows supports this via a system called WoW64, which is simply a compatibility layer on 64-bit for running 32-bit applications. However, the data structures necessary to interpret a process's memory layout is different depending on whether it is a 32-bit or 64-bit process, and the engineering effort in getting this VMI implementation correct seems to be the main hurdle.

We are also looking to extend the range of operating systems supported by Minerva in the near future to newer versions of Windows, but have not yet made plans to support other operating systems like Android, OS X and Linux.

Single execution context only. An interesting limitation to our approach is that we only consider malware propagation that happens within a single system execution. There are malware samples with propagation strategies that stretch over a system reboot by, for example, dropping a rootkit or a bootkit which is initialised

at system start-up. It would be interesting to investigate if we can extract more information about the malware by extending the analysis to several system boots.

4.9 Related work

System-wide malware tracing. The work closest to ours is Panorama [164] and DiskDuster [5]. Panorama is built on top of TEMU and proposes using system-wide dynamic taint analysis for malware analysis and detection. Panorama implements the core system that performs system-wide tainting and also several techniques that use the tainting in concrete malware analysis contexts. Although the primary malware analysis in Panorama is a keylogger detector, the authors also suggest the idea of using dynamic taint analysis to trace malware execution. However, they do not explain the idea in detail and they do not evaluate this aspect of the tool.

DiskDuster is a tool for automatic analysis and recovery of intrusions [5]. It is based on a record-and-replay system where the idea is to apply lightweight recording during execution, and in the event of an attack or infection, the recording can be replayed with heavy analysis to recover the attacked system by identifying files on disk that are either malicious or suspicious. During the replay, DiskDuster detects attacks by tainting all incoming network data to monitor for control-flow hijacking attacks and using AV scanners to identify malicious applications. Whenever DiskDuster detects an attack of either type, DiskDuster considers the process in which the attack occurred as malicious and will from that point on track the given process via DiskDuster's process monitor. For each malicious process, DiskDuster's process monitor applies a variety of rules on how to track and treat the process. Two of these rules are that every write performed by a malicious process will be tainted, and every process created by a malicious process will also be considered malicious. DiskDuster tracks process creations by way of hooking various functions in `kernel32.dll`. In this way, DiskDuster will maintain a set of malicious processes in order to capture the set of files on disk that are influenced by the malware.

From a high-level point of view, the main difference between our system and DiskDuster is that our goal is to enhance and automate malware analysis, where DiskDuster's goal is to automate intrusion recovery. Minerva focuses on capturing the system-wide malware propagation via a well-defined notion of malware execution trace and trace-collection algorithm and extracting intrinsic insights about the malicious code injections by way of the injection mechanics and the code injection graph. These parts are unique to Minerva. The overlap of Minerva and DiskDuster is that

they both aim to trace malware execution through the system. However, they do so in different ways. In particular, Minerva taints the specific memory that represents the malware, where DiskDuster considers entire processes malicious and taints all incoming network data. Minerva exclusively introduces new taint to the system when already-tainted instructions write data, whereas DiskDuster introduces taint for all writes from processes that are considered malicious. Furthermore, DiskDuster tracks multi-process execution by way of hooking process-creation functions and considers every process started by a malicious process as malicious too, whereas Minerva explicitly avoids this. As such, Minerva operates purely on an instruction-level abstraction and traces the malware by tainting the actual malware memory whereas DiskDuster applies various process-level abstractions, does not taint the malware code itself but rather the data that is written by malicious processes and incoming network traffic, and also monitors various functions to follow malware propagation.

Host-based code injections. In recent years there has been an increase in research on code injections in malware. Barabosch et al. [11] describe host-based code injection attacks and give a broad classification of various techniques. They distinguish between targeted and shotgun injections where the malware injects code into specific processes versus all processes on the system, respectively. In this terminology, the Tinba sample we describe that injects into all processes on the system is considered a shotgun approach, and the Gapz malware samples are considered targeted. Barabosch et al. study code injections in malware empirically via CuckooSandbox by writing behavioural-signatures on top of it and also perform manual reverse engineering of some samples. In [10] the same authors build a system to detect code-injection attacks by utilising the concept of honeypot processes, which are benign processes controlled by the defender that does integrity checks at run time to monitor for injection. Ispoglou and Payer take a different route to the study of multi-process malware [77]. They suggest multi-process execution as a means of evading dynamic analysis by partitioning a regular program into smaller partitions that execute in different processes and then use a scheduler to ensure consistency of the program state.

Memory forensics. Besides related work in the more general scope of malware analysis, digital forensics is an area that also provides solutions to the problem of identifying how malware infects a system. The general approach is to look for consistency in the code of a given memory snapshot and detect anomalies or atypical code in well-known processes. For example, the Gapz malware places shellcode

within `explorer.exe` by overwriting the function `atan` inside of `ntdll`. Code integrity checks that verify the provenance of code in memory images can identify that malware overwrote the function by analysing memory images [155]. Volatility [64] is a popular open source project used for digital forensics, and indeed there are plugins for volatility aimed at identifying code injection [9, 114]. The difference between our approach and that of memory forensics is the forensic analysis uses snapshots where our approach analyses an execution. This means if the Gapz would rewrite the `atan` function after having executed its shellcode, such that the `atan` function would contain its original instructions, and the snapshot of the memory image was taken post this rewriting, then forensics on this snapshot would not reveal the overwritten code.

4.10 Chapter summary

In this chapter, we concerned ourselves with tracing malware executions in the context of system-wide propagation and code reuse-attacks and divided the problem into two smaller tasks. First, we considered how to trace malware execution at the instruction-level in a general, complete and precise manner. Second, we considered how to identify code injections in this trace and automatically gather information about their core composition.

We outlined a formal framework to describe the problem of dynamically tracing malware execution in a precise and formal manner. We used this framework to develop a novel approach to capturing a malware execution trace that relies on information flow analysis, and then came up with several abstractions on the trace to identify code injections and derive intrinsic aspects of these.

We implemented our solutions in our Minerva system and tested them in the context of ground-truth applications that have previously reported to evade fine-grained analysis and host-based intrusion prevention systems. Our results show that Minerva accurately captures the malware propagation without prior knowledge of the injection techniques. Our comparative evaluation shows that we improve the completeness of malware tracing over the state-of-the-art, albeit at the cost of performance, and our detailed case studies show that Minerva can capture code injections precisely.

Chapter 5

Precise system-wide concatic malware unpacking

> "*It has long been my personal view that the separation of practical and theoretical work is artificial and injurious.*"
>
> — Christopher Strachey, Towards a formal semantics, 1966.

Run time packing is a common approach malware use to obfuscate their payloads, and automatic unpacking is, therefore, highly relevant. The problem has received much attention, and so far, solutions based on dynamic analysis have been the most successful. Nevertheless, existing solutions lack in several areas, both conceptually and architecturally, because they focus on a limited part of the unpacking problem. These limitations significantly impact their applicability, and current unpackers have, therefore, experienced limited adoption.

In this chapter, we present extensions to Minerva for effective automatic unpacking of malware samples. Minerva introduces a unified approach to precisely uncover execution waves in a packed malware sample and produce PE files that are well-suited for follow-up static analysis. At the core, Minerva deploys a novel information flow model of system-wide dynamically generated code, precise collection of API calls and a new approach for merging execution waves and API calls. Together, these novelties amplify the generality and precision of automatic unpacking and make the output of Minerva highly usable. We extensively evaluate Minerva against synthetic and real-world malware samples and show that our techniques significantly improve on several aspects compared to previous work.

5.1 Introduction

Conceptually, run time packers encode a binary with obfuscation techniques such as compression and encryption to harden analysis of their code. This hardening significantly increases the effort needed to reverse engineer a given sample, whether manually or automatically, because it requires inverting the anti-analysis techniques used by the packer to understand the full capabilities of the malware. Run time packing is a highly effective anti-analysis technique, and estimates show more than 80% of malware samples come packed [69]. The combination of needing to unpack samples before proper analysis is feasible, and that most malware samples come packed makes it desirable to develop approaches that automatically unpack malware.

Techniques and tools for automatic unpacking malware have received a lot of attention in the literature [52, 75, 81, 83, 91, 112, 147]. Despite this large amount of research, the vast amounts of work rely on the same core principle, the "write-then-execute" heuristic. This heuristic deploys the key observation that in order to execute the encrypted code, it first must be decrypted and, therefore, be dynamically generated. The most common approach by previous work is, therefore, to execute a given sample, monitor all memory writes made by the malware, and whenever dynamically written memory executes, the unpacker identifies this memory as decrypted. The tools then dump this specific memory to enable follow-up inspection.

There are two main limitations to the approach of existing work. First, the "write-then-execute" heuristic is not well-suited for packers that perform system-wide unpacking. This is because the heuristic only captures code that is dynamically generated *explicitly* by the malware and not malicious code that is dynamically generated *via* benign code which, unfortunately, is frequently the case in multi-process unpacking. Consequently, existing unpackers are mainly suitable for single-process malware and new approaches to capture system-wide malware unpacking are needed. Second, the primary output of existing work is memory dumps or naively constructed PE files of the dynamically generated code. The output lacks structure, is often an unreasonable over- or under-approximation of the actual malware code, and many obfuscation techniques from the packing process, e.g. obfuscation of external dependencies, remain in the output. As such, the analysis that follows must overcome these obfuscation techniques to enable meaningful analysis of the code. This is a problem because the purpose of unpacking is to facilitate follow-up analysis and not to give any conclusive answer about the malware itself.

The limitations described above reoccur in existing work, and we argue that an essential reason for this is because existing work widely uses the same set of benchmark applications to validate their solutions. These benchmark applications consist of packers that are out-dated and built more than a decade ago. Consequently, the empirical assessment of novel tools occur with old, and often similar, techniques that do not accurately reflect the challenges posed by modern-day malware packers. To ensure that our novel tools are relevant, we need new benchmark applications that can be used for profiling novel unpackers. These benchmarks must explore corner-cases of modern packing techniques and be easily accessible to anti-malware researchers.

The goal of this chapter is to develop techniques on top of Minerva to overcome the limitations of existing work highlighted above. We present a unified approach to precisely unpack malware samples with system-wide execution, dynamically generated code, custom IAT loading and API call obfuscations. The aim is to provide unpacked code that is well-suited for follow-up analysis via manual reverse engineering or off-the-shelf static analysis tools. To this end, Minerva deploys a *combination* of dynamic and static analysis to amplify the effectiveness of automatic unpacking. The novel techniques presented in Minerva rely on information flow, which makes it highly precise and capable of unpacking malware samples in a system-wide context. Minerva models execution waves on a per-process basis and each process with malware execution operate within the context of a single execution wave at any given moment. This provides for a clear wave model and implementation but may result in duplicate content amongst waves, for example, when execution waves use code from an earlier execution wave.

Minerva takes as input a 32-bit Windows binary and outputs at least one Portable Executable (PE) file per execution wave. This has the benefit of mostly independent PE files but also means the duplicate content of multiple waves will exist in multiple PE files. In order to produce output that is useful for follow-up analysis, Minerva captures how the malware uses external dependencies throughout the entire execution and maps this to each execution wave, resulting in PE files with valid import address tables and patched API calls. Finally, Minerva also performs static analysis to identify relevant malware code within each execution wave. In addition to our unpacker, we also propose a new benchmark suite with applications that combine code-injection techniques, dynamically generated code and obfuscation of external dependencies to overcome the limitations of empirical evaluation in existing work. We demonstrate our unpacker empirically against synthetic and real-world malware samples.

Our main contributions of this chapter are as follows.

- We present a novel approach that combines dynamic and static analysis techniques to unpack malware that executes across the entire system automatically. The approach focuses on precise analysis and outputs unpacked samples that are well-suited for follow-up static analysis.
- We present a new benchmark suite with samples exploring modern-day packing behaviours. To the knowledge of the author, this is the first benchmark suite that comprises synthetic applications aimed at evaluating unpackers.
- We implement the techniques into Minerva and present an extensive empirical evaluation based on synthetic applications and real-world malware samples.

5.2 Background, motivation and overview

Packing is an umbrella term that refers to a set of various concrete obfuscation techniques and there is no clear definition on the specific obfuscation techniques it encapsulates. This section clarifies the obfuscation techniques we treat in this chapter and the limitations of existing work that motivate us. In total, we have compiled six core limitations across two general obfuscation techniques.

5.2.1 Dynamically generated code

The obfuscation technique that is most commonly associated with packing is dynamically generated code. In its simplest terms, dynamically generated code is when an application writes memory at run time and then proceeds to execute this memory. Most often malware does this by containing encrypted code inside its binary image and decrypting this at run time in order to execute it. Existing automated unpackers identify dynamically generated code with the write-then-execute heuristic. This heuristic partitions the malware execution into a set of layers $\mathcal{L}_0$, $\mathcal{L}_1$, ... $\mathcal{L}_n$ such that each layer constitutes dynamically generated code. Layer $\mathcal{L}_0$ represents the instructions of the binary malware image when first loaded into memory and $\mathcal{L}_{i+1}$ represents the instructions executed on memory written by the instructions in layer $\mathcal{L}_i$.

Limitation 1.1: the write-then-execute heuristic is unable to capture dynamically generated malicious code *via* benign code. The strict relationship that instructions of one layer must be dynamically generated explicitly by instructions from a previous layer severely limits the generality of existing work. Malware that uses benign code to dynamically generate its malicious code go unnoticed by this

model. The implications of this limitation are substantial for capturing dynamically generated code across multiple processes by way of code-reuse attacks or OS-provided APIs since it is not the instructions of the malicious code that does the writing of memory. Rather, it is benign code that is manipulated by the malware into writing dynamically generated malicious code.

The Gapz malware introduced in Section 4.2 is an example of this where we find the most recent unpacker by Ugarte et al. [147] unable to identify the code in the `explorer.exe` process. This is because Gapz uses code-reuse attacks within `explorer.exe` to dynamically generate malicious code, also inside `explorer.exe`. As such, the malware forces benign code to dynamically generate malicious code on its behalf, which is not captured by the write-then-execute model. The work by Ugarte et al. only reports one process used in each of the Gapz samples that use benign code to propagate its malicious code, missing all injections into `explorer.exe`, and the result is similar for other tools that rely on the write-then-execute heuristic [21, 52].

Limitation 1.2: existing work unreasonably approximate relevant dynamically generated memory. Whenever an unpacker observes dynamically generated code it outputs the code for follow-up analysis. To do this, the unpacker must have a definition of what parts of dynamically generated memory are relevant to the unpacked code. This is because not all memory that is dynamically generated, e.g. the stack, is relevant for the unpacked output. However, this step of identifying relevant memory is highly overlooked by previous work. For example, neither Renovo [83] nor EtherUnpack [52] clearly describe the specific memory they extract during unpacking, and Mutant-X [75] dumps the entire memory image of a process when observing dynamically generated code. These are unreasonably imprecise and leave follow-up analysis with the task of identifying a needle in a haystack.

Limitation 1.3: existing work output raw memory scattered across many memory dumps. The majority of existing unpackers [21, 52, 83, 147] make little effort to output the unpacked code in a coherent data structure but rather output the unpacked malware in the shape of raw memory dumps. The problem is that when malware dynamically generates code, this may be scattered across several regions, and some of these may also be data-only sections. A precise unpacker should not output incoherent raw memory regions, but rather a suitable data structure that combines these memory regions in an appropriate manner, e.g. re-basing

where needed, that enables meaningful follow-up analysis.

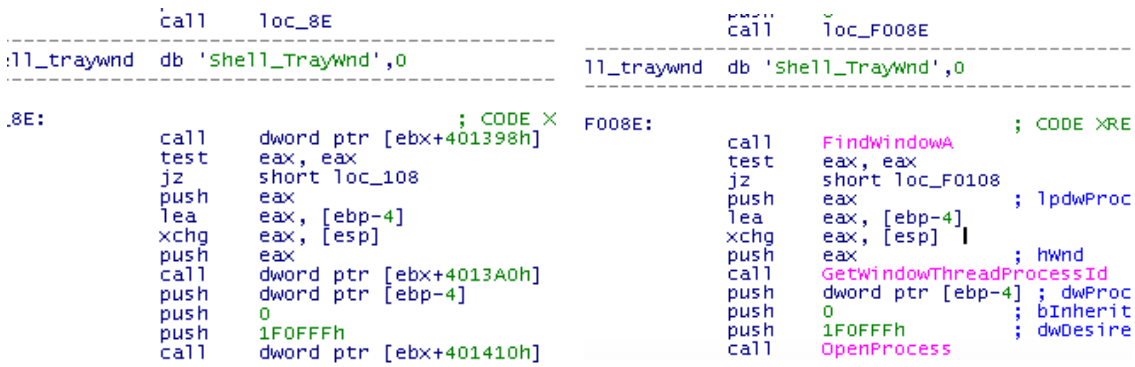
5.2.2 Obfuscating external dependencies

The way malware interacts with its environment is significant to understanding its malicious activities. We capture this understanding by analysing how the malware uses the OS via API calls and system calls, and these are, therefore, natural obfuscation targets for malware.

Limitation 2.1: existing unpackers fail to accurately correlate API calls to malicious code. There is often a large portion of dynamically generated code that must be covered when analysing packed malware. To quickly navigate towards relevant parts, we rely on the malware's use of APIs. However, existing unpackers fail to accurately correlate API calls within a process to the packed code, or even, more generally, attribute whether a given API call was performed by malicious or benign code. Consequently, they report unreasonable estimates of API-usage by the malware.

In order to circumvent this limitation, the unpacker needs to maintain knowledge of which code belongs to the malware and also be able to identify the instruction responsible for a given API call. From an engineering point of view, most unpackers will be able to augment their systems with solutions to these problems with a modest implementation effort. However, fundamentally, this limitation is guarded by the ability to correctly identify what code belongs to the packed malware, which is closely related to Limitation 1.1, Limitation 1.2 and also the problems discussed in the previous chapter. We identify it here because it is an essential feature in terms of understanding how the unpacked malware code uses external dependencies and something that current unpackers do not support. For example, when the unpacker from Ugarte et al. is matched with a sample[1] from the Tinba malware family that creates one layer of dynamically generated code before injecting code into the Windows process `winver.exe`, the unpacker reports that the dynamically generated code performs 1666 API calls from more than 350 different API functions. This result far exceeds the correct count, which is fourteen API calls from ten different API functions.

[1] sha256 078a122a9401dd47a61369ac769d9e707d9e86bdf7ad91708510b9a4584e8d4



(a) Traditional unpacker, from [147]

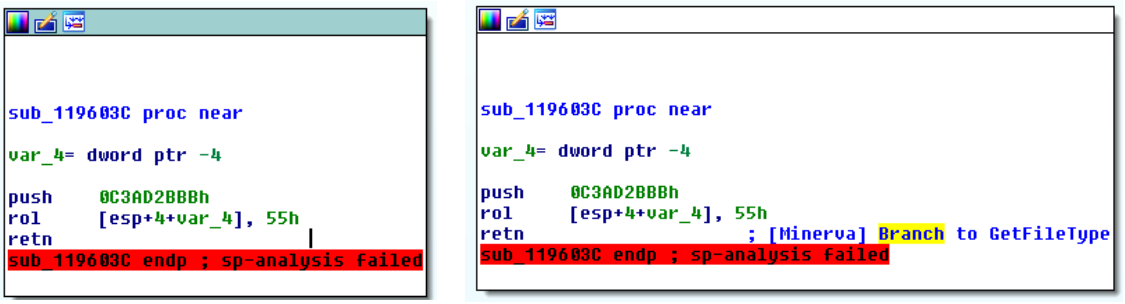
(b) Minerva

Figure 5.1: The output of unpackers when being matched with API calls that are obfuscated with custom API resolution and that branch via a temporal register value.

Limitation 2.2: output from existing unpackers do not show API-usage when faced with custom API-call resolution. There is an intricate relationship between dynamically generated code and API call obfuscation. In regular PE binaries, the import address table (IAT) specify the external modules the given application uses. At run time, the operating system linker uses this IAT to load these modules and resolve the addresses of the specific functions the binary imports. However, packed code minimises the IAT to hide how it uses external dependencies, and, instead of using the regular OS linker, the packer deploys a custom linker to resolve its imports.

Custom API resolution can happen at any moment(s) during execution and memory dumps taken by existing unpackers are, therefore, susceptible to occur when the malware is yet to resolve its external dependencies. Unfortunately, it is rare that API resolution has occurred the moment dynamically generated code is observed, which is precisely when existing work dumps the memory [21, 52, 83]. Figure 5.1 shows the differences of matching a traditional unpacker (Figure 5.1a) with a sample that has custom API resolution and matching the same sample with an unpacker that accurately captures API-usage in unpacked code (Figure 5.1b), in this case, a result of Minerva. It is clear that without knowledge of the API calls it is hopeless to determine the activities of the code, whereas it is clear from the Minerva-generated code.

Limitation 2.3: existing unpackers are unable to identify obfuscated API calls. Orthogonal to the resolution of API calls, some packed malware samples will go a step further and directly obfuscate the way they call external APIs. In general, there are many ways for malware to obfuscate API calls. In Figure 5.1a,



(a) Traditional unpacker (b) Minerva

Figure 5.2: The output of unpackers when being matched with an API obfuscation from the PEtite packer.

we see that the raw calls from Tinba depend on the value of `EBX`, which in this case contains the base-offset of a custom IAT by the malware. Furthermore, Figure 5.2 shows an example from an application packed with the PEtite[2] packer where the code calls a Windows API function by pushing a value on top of the stack, rotating that value and then transferring execution via a `ret` instruction to the rotated value on top of the stack.

The output of existing unpackers is not capable of resolving obfuscated API calls in the unpacked code. This is a problem because it is much harder and sometimes impossible, to determine the destination of the branch instructions in follow-up analysis than it is for the unpacker. For example, without knowledge about the contents of `EBX`, the data at the address being read and the process layout, it is impossible to determine the destination of the given branch instructions and if they are API calls.

5.2.3 Solution overview

The goal of this chapter is to develop system-wide, precise and general unpacking techniques into Minerva. Specifically, our goal is to input a malware binary in Minerva and output PE files that precisely capture the malware code post-decryption and decompression, and also capture how the malware uses external dependencies. The aim is to output PE files that are well-suited for follow-up analysis by off-the-shelf static analysis tools and manual investigation.

To achieve our goal, we must overcome the limitations highlighted above. First, to overcome the limitations when dealing with dynamically generated code, we need a solution that can (*limitation 1.1*) identify dynamically generated memory across the system; (*limitation 1.2*) extract *precisely* the memory that is relevant to the malware;

[2] `https://www.un4seen.com/petite/`

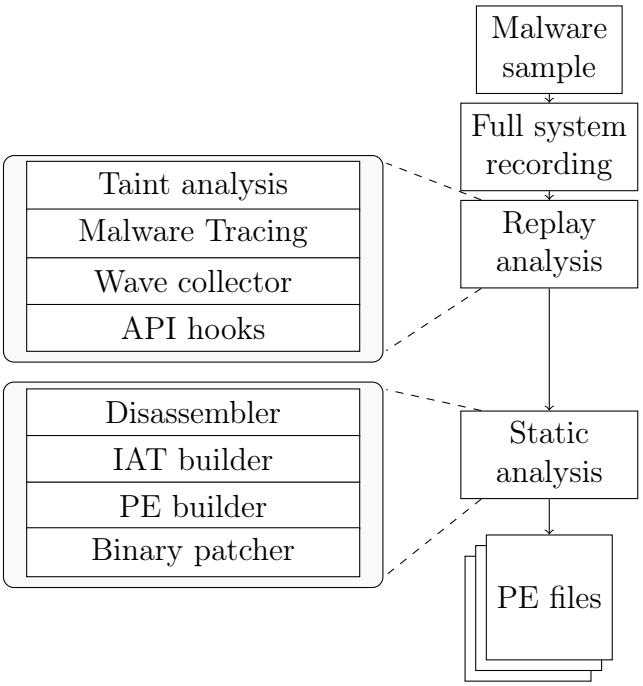


Figure 5.3: Architecture of Minerva's automatic unpacker.

and (*limitation 1.3*) combine the relevant dynamically generated code into meaningful and related structures. Second, to overcome the limitations against malware that obfuscates external dependencies, the solution must also (*limitation 2.1*) precisely capture the use of API calls *within* the malware code; (*limitation 2.2 and 2.3*) do this in the context of custom API resolution and obfuscated API calls; and, finally, map these observations to the output, so it is readily available for follow-up analysis.

The solution we come up with, and implement into Minerva, deploys a two-step approach following the architecture shown in Figure 5.3. First, we use the dynamic analysis in Minerva to precisely extract packed code and the API calls of the malware, and then we use static analysis to construct PE files based on the unpacked code. Specifically, the first step abstracts the malware execution trace into execution waves based on information flow analysis. An execution wave is a process-level construct that represents dynamically generated code in the malware. During the run time analysis, Minerva also ensures precise identification of API calls by the instructions in the malware execution trace. From the first step, we get a set of execution waves consisting of memory dumps, the malware execution trace of each wave and more. The second step Minerva performs is to group related memory within each execution wave using disassembly techniques. Minerva then converts each group of related dumps into a new PE file with a new import address table, and patches API calls based on static analysis and the API calls observed during dynamic analysis. In the

following sections, we detail these steps.

5.3 Information flow execution waves

Given the malware execution trace, the first step is to partition it into execution waves. The goal of execution waves is to capture dynamically generated malicious code independently of who wrote the code and on the basis that the generated code must originate from the malware. However, we consider execution waves to be more than just a sequence of instructions. The set of execution waves gives an explicit representation of an entire application, including dynamically generated malicious code, and each execution wave may, therefore, include both executable and non-executable data.

In this section we give a semantics for execution waves (Section 5.3.1) and describe how we collect the waves in practice (Section 5.3.2).

5.3.1 Execution wave semantics

The goal of our execution wave semantics is to clearly define the conversion of a malware sample's execution into waves of dynamically generated malicious code. As such, we describe the waves in relation to an execution trace $T(P, E)$ described in Section 4.3.

We partition the malware execution into waves on a process-level basis. We map every instruction in the malware execution trace $i \in \Pi_A$ to a process P_y and a wave within this process W_x. We denote $P_y W_x$ to mean wave x within process y, and every process with malicious code execution contains a sequence of waves $P_y.\Omega = P_y W_0, \ldots, P_y W_n$ with $|P_y.\Omega| \geqslant 1$. We denote the initial wave in which malware execution begins as $P_\epsilon W_\epsilon$ and the set Φ_Π contains all execution waves for a given malware execution trace Π_A. For each instruction in the malware execution trace, we first identify the process in which they execute and then the wave they belong to within their respective process.

Formally, we define an execution wave as follows.

Definition 2. *An execution wave is a tuple composed of:*

- *A sequence of instructions $\mathcal{I} = i_0, \ldots, i_n$ executed in the given wave. We have i_0 to be the entry point of the wave;*
- *a shadow memory $\mathcal{S}$, which is a set of ordered pairs (m_{addr}, m_{byte}) that contains the tainted memory making up the wave, including both code and data memory;*

- *the tainted writes $\mathcal{T}$ which is a set of ordered pairs (t_{addr}, t_{byte}) that holds the tainted memory written by instructions in P since i_0, where P is the process of the execution wave.*

Next, we present a set that formalises our requirements for partitioning a complete execution trace into a set of execution waves. The purpose of this definition is to capture every layer of dynamically generated malicious code and not restrict a minimal overlap between the content of each execution wave. In the following, we write for two instructions i, j, $i < j$ if i comes before j in the malware execution trace, and vice versa.

Definition 3. *Let T(P,E) be an instruction execution trace and Π_A the corresponding malware execution trace. The set of execution waves is then given $\Phi_\Pi = \{P_0, \ldots, P_n\}$ where:*

- $\forall i \in \Pi_A \exists P_x \in \Phi_\Pi | i \in P_x.\mathcal{I}$.
- *For any P_yW_x and P_yW_z in Φ_Π where $x < z$ we have that $\forall i_x \in P_yW_x.\mathcal{I}, \forall i_z \in P_yW_z.\mathcal{I} | i_x < i_z$.*

  This says that there is a strict ordering in the malware execution trace between the instructions of any two waves in a given process $P_y.\Omega$.
- $\forall (m_{addr}, m_{byte}) \in P_wW_{w'}.\mathcal{S} \exists (t_{addr}, t_{byte}) \in P_tW_{t'}.\mathcal{T} | (m_{addr}, m_{byte}) \in P_\epsilon W_\epsilon.\mathcal{S} \vee (m_{addr}, m_{byte}) = (t_{addr}, t_{byte})$ *where* $P_wW_{w'} \neq P_tW_{t'}$ *and* $\forall i_w \in P_wW_{w'}.\mathcal{I} \exists i_t \in P_tW_{t'} | i_t < i_w$.

  This says that the shadow memory for all execution waves must either exist in the shadow memory of the initial wave or be composed of tainted memory written by a wave that started earlier.
- *For any wave $P_yW_x \in \Phi_\Pi$ we have that $\forall i \in P_yW_x.\mathcal{I} | \exists (m_{addr}, m_{byte}) \in P_yW_x.\mathcal{S} | i[A] = m_{addr}$.*

  This says the memory of any instruction in each execution wave must be present in the shadow memory of the given wave.

An important aspect of Definition 3 is that the second bullet enforces a strict ordering between instructions in the set of execution waves for each process. The effect of this is that we preclude instructions from any given execution wave to be used in any other execution wave. The reason we do this is that it creates a clear

history of execution wave progress within each process, and it becomes easier to implement since it is only necessary to maintain one execution wave per process. The drawback is that when malware transfers execution to code from an earlier execution wave, we include some content of the earlier execution wave into the current execution wave. In this way, we may end up with waves that overlap in their shadow memory, but, naturally, this can be stripped during post-processing. However, we have found this to be no major issue and that the trade-off works well in practice. We leave the door open and encourage future work in other models, e.g. more refined models, in Section 7.2.

5.3.2 Collecting the execution waves

In practice, we only associate one wave with a given process at any given moment. Therefore, to collect the execution waves, it is sufficient to keep track of the current wave in each process. We initially only have one, wave which is the wave inside of the process executing the malicious application. The shadow memory $\mathcal{S}$ of this wave is the malware module when loaded into memory, and the set of tainted writes is initially the empty set, $\mathcal{T} = \varnothing$. We then update the set of tainted writes whenever an instruction writes tainted memory following our revised Update function shown in Algorithm 4.

To capture execution waves, we monitor for each process the relationship between the currently executing instruction, the shadow memory and the set of tainted writes following Algorithm 3. Specifically, for every tainted instruction in the malware execution trace, there are four possible cases:

1. The address of the instruction is not in the shadow memory and not in the tainted writes (line 11 Algorithm 3);

2. The address of the instruction is not in the shadow memory but in the tainted writes (line 14 Algorithm 3);

3. The address of the instruction is in the shadow memory and in the tainted writes but the content of the shadow memory is not similar to current instruction (line 17 Algorithm 3);

4. The address of the instruction is in the shadow memory and in the tainted writes and the content of the shadow memory is equivalent to the memory of the current instruction (line 19 Algorithm 3).

ALGORITHM 3: Wave collection

Data: (input) Malware sample B
Result: Logged malware execution waves and malware execution trace Π.

```
1  P ← init_taint(B)
2  T, S ←init_waves(B)                    // initialise the shadow memories and tainted writes.
3  // Full system instrumentation
4  i ← first_instr()
5  while P ≠ ∅ do
6      // is the instruction tainted?
7      if i[A] ∈ P then
8          Π ← Π^⟨i⟩
9          if i[A] ∉ S_pid then
10             if i[A] ∉ T_pid then
11                 S_pid ← S_pid ∪ (i[A], i[mem])
12                 W_pid ← W_pid ∪ {i}
13             else
14                 S_pid, T_pid, W_pid ← dump_wave()
15         else
16             if i[A] ∈ T_pid ∧ S_pid[i[A]] ≠ i[mem] then
17                 S_pid, T_pid, W_pid ← dump_wave()
18             else
19                 W_pid ← W_pid ∪ {i}
20     i, P, T = update(i, P, T)
21 return (Π)
```

Case (1) happens in two scenarios. The first case is when tainted memory is transferred across processes via shared memory. For example, if tainted memory is written to memory shared by processes P_1 and P_2 and the instructions performing the writing is in P_1, then the tainted writes will not be in $P_2.\mathcal{T}$ or $P_2.\mathcal{S}$ because we only populate $P_2.\mathcal{T}$ if instructions from $P_2.\mathcal{T}$ are writing to the address space of P_2. The second case is when code from the current wave transfers execution to code that is part of an earlier wave. This is because the shadow memory of each wave does not propagate to the proceeding wave, but the memory remains tainted nonetheless. Whenever we observe case (1) we add the memory of the instruction to the shadow memory of the current process and also append the instruction to the sequence of instructions in the current wave. In this case, we update the shadow memory of the current wave with the executing instruction.

In case (4) the current instruction is simply part of the current execution wave, and this is by far the most common case. In this case, we append the instruction to the instruction sequence of the current wave. In cases (2) and (3) we consider the current instruction to be the entry point of a new execution wave. Specifically, in case (2) the instruction is dynamically generated in a new memory region and in case (3) the instruction is dynamically generated on top of already existing malware code.

In the event of a new wave, we log information about the current wave following Algorithm 5. First, we log the instructions executed in the current wave, tainted writes and the shadow memory, which includes dumping every page in which there is a tainted write and also dumping the shadow memory. Then we set the shadow memory of the next wave to be the tainted writes of the current wave and set the tainted writes to be the empty set.

ALGORITHM 4: Update

Data: (input)Instruction i, memory propagation set P, Tainted writes T.
Result: Next instruction i_{next}, memory propagation set P, Tainted writes T

$\mathcal{P} \leftarrow propagate\_taint(i, \mathcal{P})$
$i_{next} \leftarrow exec\_instr(i)$
if $i[A] \in \mathcal{P}$ **then**
  for $o \in i[O]$ **do**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{o\}$
for $w \in i[W]$ **do**
  if $w \in \mathcal{P}$ **then** $\mathcal{T}[i.pid] \leftarrow \mathcal{T}[i.pid] \cup \{w\}$
return $i_{next}, \mathcal{P}, \mathcal{T}$

ALGORITHM 5: dump_wave

Data: (input)Current wave $\mathcal{W}$, shadow memories $\mathcal{S}$, Tainted writes $\mathcal{T}$.
Result: Updated $\mathcal{S}$, $\mathcal{T}$, $\mathcal{W}$

LogInstrs(W_{pid})
LogTaint($\mathcal{T}$)
LogShadowMem($\mathcal{S}$)
$\mathcal{S}_{pid} \leftarrow \mathcal{T}_{pid}$
$\mathcal{T}_{pid} \leftarrow \varnothing$
$W_{pid} = \varnothing \cup \{i\}$
return $\mathcal{S}_{pid}, \mathcal{T}_{pid}, W_{pid}$

The execution waves capture dynamically generated code independent of who wrote the code including dynamically generated *malicious* code via benign code. We achieve this generality because the shadow table is composed of tainted memory and tainted memory propagates through both benign and malicious instructions. Since the tainted code originates from the malware itself, it is dynamically generated *malicious* code. This property distinguishes our technique from previous work and allows it to be more general without losing precision.

The output from collecting the execution waves is the sequence of waves executed during the malware execution. For each execution wave, we have memory dumps of the tainted memory during its execution and the list of instructions that belong to each wave. As such, we have an explicit representation of each instruction in the malware execution in the form of its raw bytes, and we also have memory dumps of

any non-executed malicious (tainted) memory. All of this information will then be used to reconstruct PE files that are effective for follow-up static analysis.

5.4 Precise dependency capture

In order for the PE files to be useful for follow-up static analysis, they must show how the malware uses external dependencies. As described in Section 5.2.2, we must consider custom API call resolution and obfuscated API calls. To this end, we capture the destination of every branch instruction in the malware execution trace and check if it corresponds to the beginning of a function in an external module.

To collect the addresses of functions in each process with malware execution, we iterate the export table of every module in the given process and capture the address of every function it exports. We put these functions in a per-process map that pairs function addresses with their respective function names. Minerva also comes with the possibility to speed up this process using pre-calculated function offsets for a given DLL. As such, with pre-calculated offsets, we only need to know the base address of a given imported module inside the malware process to compute the absolute addresses of its exported functions.

To capture the API functions that the malware calls, we obtain the destination of every branching instruction in the malware execution trace. If the branch destination is in the set of functions exported by any of the dynamically loaded modules within the execution trace, it means the malware performs an API call. We log every API call and for some functions the parameters as well. For many functions in the Windows API, the return value is also essential to understand the semantics of the call. To capture the return value and output parameters, we note the return address of the API call on the stack and read the output of the function whenever the return address executes. We also monitor functions like `LoadLibrary` to update our export table when processes load new modules.

Our approach to monitoring API calls precisely captures the API calls performed by instructions in the malware execution trace and do not capture API calls performed by benign code inside a process in which the malware executes. Furthermore, because we know the specific malicious instruction for each execution wave, it is trivial to map API calls to execution waves. This precise mapping highly improves the precision of the analysis in comparison to sandboxes that capture API calls globally within a process since many of these calls are irrelevant to the malware (this is particularly true in code injected processes).

Minerva currently does not take any efforts when malware hides API usage by way of stolen bytes or copying of the Windows code. Furthermore, if malware deploys inlined library code or statically linked libraries, then Minerva will not consider these as external dependencies. This is a limitation we discuss further in Section 5.7.

5.5 Static reconstruction of execution waves

After collecting the execution waves and external dependencies, we need to combine these into PE files. For each execution wave we construct a set of PE files based on the content of their respective shadow memory and for each PE file we need three ingredients: (1) The specific memory pages of an execution wave that makes up the PE file; (2) the PE's IAT; (3) the entry point of the file.

The static analysis component of Minerva performs three main steps. First, it groups related memory dumps of each execution wave, then identifies external dependencies in each of these memory groups and, finally, builds new PE files based on the results of the two previous steps.

5.5.1 Merging over-approximated shadow memories

The output from collecting the execution waves includes, for each execution wave, page-level memory dumps of the shadow memory and tainted writes. Intuitively, it can seem appropriate to convert all of these memory dumps into one large PE file and use this for static analysis. However, we have found this to be imprecise in practice because it is rarely the case that all of the memory dumps are relevant to the malware. On the one hand, the shadow memory is a conservative approximation as we capture some memory that is not executable code, and some memory is a result of over-propagated taint. On the other hand, we do not want to reconstruct PE files purely based on executed memory since this will miss non-executed, yet still, malicious executable code, and also relevant data sections.

To avoid this imprecision, we divide the page-level memory dumps from the dynamic analysis into smaller groups, such that the pages of each group are related and no page in a given partition relates to any other partition. The goal with this is to capture the parts of the malware that are self-contained and represent the application timelessly. To this end, we create a PE file with multiple sections for each partition. Figure 5.4 shows an example of how we select the specific tainted pages that are relevant for the unpacked malware from a set of tainted pages output by Minerva's dynamic analysis component.

The first step is to identify the tainted pages with malicious code execution. To do this, we first iterate the sequence of instructions executed in a given execution wave and collect all pages that hold instructions from this sequence. Following this, we iteratively collect neighbouring pages until there are no more neighbouring pages and the result is a set of page-level intervals where some pages hold executed code, and other pages neighbour up to these. This corresponds to the first two steps in Figure 5.4.

Following this, we identify pages in the shadow memory that relate to each interval. To construct self-contained PE files, we capture data-dependencies and control-dependencies to other pages in the shadow memory for each interval. We do this by performing speculative disassembly on each memory dump to capture cross-references to other memory dumps. This step gives us cross-references for each interval, and we then iteratively merge related intervals such that no interval will have cross-references to other intervals. Following this approach, we end up with a set of groups of memory dumps, and we create a PE file for each of these groups. In the example in Figure 5.4 we end up with one group consisting of two intervals and will, therefore, create one PE file.

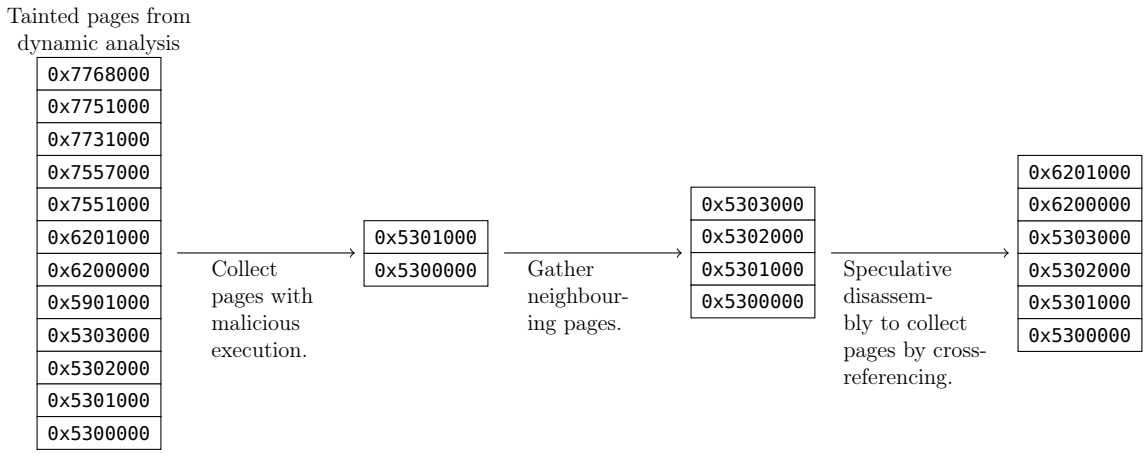


Figure 5.4: The process of identifying which tainted pages from dynamic analysis that are relevant when reconstructing unpacked PE files. In the example, the reconstructed PE file has two sections (`0x5300000-0x5303000` and `0x6200000-0x6201000`).

5.5.2 Dependency reconstruction

To reconstruct external dependencies in our PE files, we need to rebuild the IAT of the binary and patch instructions to rely on this new IAT.

To construct the IAT, we first identify API calls made by instructions belonging to the pages of each memory group. We identify these by matching the API hooks collected during dynamic analysis to instructions of the respective code wave and the pages of the given memory group. We include each unique API function in the IAT of the reconstructed PE file.

Although we know which instructions branch to external APIs from the malware execution trace, the branch destinations may not be visible from the memory dumps themselves. The final step in constructing PE files is, therefore, to map the instructions that perform API calls to our newly generated IAT by patching them on the binary level. Unfortunately, binary patching is not an easy task since some instructions may require for us to rearrange the instructions in the binary, and this may subsequently break it. In practice, we patch branch instructions that are 6 bytes long, e.g. `call [0xdeadbeef]`, because we can do this without rearranging instructions. We do not patch instructions that are less than 6 bytes, e.g. `call eax`. However, we still keep the cross-references so they can be used in more abstract representations in a follow-up analysis, for example, Minerva comes with a script that inserts cross-references and appropriate comments to an IDA Pro database of the unpacked malware.

5.5.3 Final PE construction

In order to construct the final PE file, we need to know the entry point and the PE sections to put in the file. To identify the entry point, we go through the instruction sequence of the given wave and identify the first instruction in the range of each memory dump group. In order to construct the PE sections, we rely on the memory intervals that we end up with in each memory group. For example, in our example from Figure 5.4 we end up with one group and two intervals ([`0x5300000-0x5303000`], [`0x6200000-0x6201000`]). We make each of these intervals into an individual section of the PE file and place the newly generated IAT in-between the PE header and these sections. The reason we make each of them into individual sections is to avoid rebasing each interval. The pages we dump from virtual memory are placed at various locations, and each of them must keep this virtual address in the PE file. As such, the `PointerToRawData` and `VirtualAddress` values in each section header will be significantly different, and the `VirtualAddress` points to the base address of each interval as it was when dumped from virtual memory (`0x5300000` and `0x6200000` in the example in Figure 5.4).

5.6 Evaluation

Having presented the core techniques of Minerva's unpacker, we now move on to evaluate Minerva using multiple benchmarks with respect to the following research questions:

1. Does Minerva precisely capture dynamically generated code and the malware's API calls?

2. Does Minerva improve results over previous work?

3. Is Minerva relevant for common malware analysis tasks?

To facilitate the research questions above, we gather four sets of benchmark applications comprising synthetic applications as well as real-world malware applications:

1. **Benchmark #1 : Ground truth data set.** We develop a new benchmark suite that combines the use of dynamically generated code, code injection and obfuscation of external dependencies. In total, we have developed nine different applications, and they are all described in Table 5.2.

   The applications in the benchmark suite represent many of the challenges posed by real-world packers. To the knowledge of the authors, this is the first dedicated benchmark suite for challenging the attributes of unpackers where none of the samples relies on packers developed by third-party teams. The benefit of this benchmark suite is that each sample poses specific challenges that are clearly defined, the applications are easy to understand, and we have the complete source code of each example. As such, it becomes much more accessible to determine if an unpacker is successful because there is no need to reverse engineer large amounts of binary code.

2. **Benchmark #2 : Selected malware samples.** The second data set corresponds to several of the malware samples also introduced in Chapter 4. These samples perform many of the obfuscation techniques that Minerva aims to overcome, such as code injection combined with dynamically generated code and custom API resolution.

3. **Benchmark #3 : Packed synthetic samples.** We have taken a set of synthetic samples and packed them with well-known packers. In these applications,

Artemis	CTBLocker	Cerber	CoinMiner
CosmicDuke	Emotet	Kovter	Madangel
Mira	Natas	Nymaim	Pony
Shifu	Simda	TinyBanker	Urausy
Zbot			

Table 5.1: The malware families in Benchmark set #4. We collected a total of seven samples from each family.

we know the applications' behaviours before packing because we design the applications; however, we do not know the exact changes the packers make on the code and, therefore, do not have ground truth about the packed applications.

4. **Benchmark #4 : Real-world malware samples.** This set comprises 119 malware samples from the real-world malware families listed in Table 5.1. We collected seven samples from each family to maintain a balanced data set, and the samples were collected from VirusTotal. In order to ensure the samples are indeed benign, we required each sample to be detected by at least 15 anti-malware vendors. Furthermore, in order to ensure certainty that the samples belong to their respective families, we required at least two vendors to label them in the same family. On average each sample had 52 anti-malware vendors report it as malicious and a median of 54. We recorded each of these samples for 25 seconds and set a max replay time of 120 minutes.

In order to assess the techniques of Minerva, we must make a fair and meaningful comparison to existing work. One approach is to compare Minerva to recently proposed unpackers like Codisasm [21] or Aranchino [122]. However, we already showed in Chapter 4 that Codisasm is very limited due to its implementation in PIN, and Aranchino is also developed on top of PIN with no additional effort for analysing system-wide malware. Instead, we compare Minerva to the unpacker by Ugarte et al. [147].

Ugarte et al. [147] propose a malware unpacker that is capable of analysing multi-process malware is implemented on top of QEMU. The unpacker supports multi-process unpacking by monitor various system calls and also develop techniques for capturing memory mappings. Unfortunately, the tool they present is only available as a web service[3], which forces us to treat their system as a black box. Furthermore, they do not mention which OS they support in their work; however, from experimenting with the service, we conclude the analysis environment is Windows XP. We

[3]www.packerinspector.com

ID	**Description**.
D1	Dynamically generates code and uses custom IAT resolution to resolve `GetModuleHandle`, `GetProcAddress` and `ExitProcess` and exits.
D2	Dynamically generates code and uses custom IAT resolution to resolve `GetModuleHandle`, `GetProcAddress` and `MessageBoxA` and then displays a message box.
D3	Dynamically generates code that further dynamically generates code and then uses custom IAT resolution to resolve `GetModuleHandle`, `GetProcAddress` and `MessageBoxA` and then displays a message box.
D4	Dynamically generates code that further dynamically generates code and then uses custom IAT resolution to resolve `GetModuleHandle`, `GetProcAddress` and `ExitProcess` and then exits.
C1	Opens the Windows process `explorer.exe` using `OpenProcess, WriteProcessMemory` and `CreateRemoteThread`, then inside the target process dynamically resolves the address of `GetModuleHandle`, `GetProcAddress` and `ExitProcess`, and calls each of them to exit.
C2	Opens the Windows process `explorer.exe` using `OpenProcess, WriteProcessMemory` and `QueueUserAPC`, then inside the target process dynamically resolves the address of `GetModuleHandle`, `GetProcAddress` and `ExitProcess`, and calls each of them to exit.
C4	Uses the PowerLoaderEx injection that relies on a global memory buffer and code-reuse attacks to hijack execution of `explorer.exe`. Inside `explorer.exe` code-reuse attacks transfers execution to shellcode that calls `LoadLibraryA`.
C5	Uses the Atombombing injection techniques that relies on the global atom tables to execute within `explorer.exe`. Inside `explorer.exe` it uses code-reuses attack to execute a piece of shellcode that launches `calc.exe`.
M1	Injects code into `explorer.exe` similarly to A1, then inside the target process dynamically generates code that then dynamically resolves the address of `GetModuleHandle`, `GetProcAddress` and `ExitProcess`, and calls each of them to exit.

Table 5.2: Description of the samples in data set #1 and how they perform code injection.

determined this because the web service responds with *"Error - The sample did not start executing."* when faced with applications compiled for Windows 7 and later, but runs normally with Windows XP applications. As such, we wrote the samples in our data set to make sure they all execute correctly on both Windows XP and Windows 7. In this chapter, we will refer to the unpacker by Ugarte et al. [147] as PackerInspector.

5.6.1 Experimental set up

We conduct all of our Minerva experiments on a 4-core Intel-7 CPU with 4.2 GHz and a Windows 7, 32-bit guest architecture. The guest is in a closed network and connected to another virtual machine that performs network simulation using INetSim[76]. As such, malware samples that connect back to some CC server will be able to resolve DNS names, connect to every IP and also receive content. However, the content itself is the default data provided by INetsim. In Table B.2 in Appendix B, we provide a checklist for prudent malware experimentation related to this chapter.

We executed the applications on the guest machine with a local admin account, and User Account Control (UAC) enabled. We perform no user stimulation during the analysis, and there were no applications apart from the generic Windows processes running in the guest machine itself.

5.6.2 Empirical evaluation of correctness

In our first experiment, we match Minerva and PackerInspector with the ground-truth samples in benchmark set #1. For each of the samples, we capture the number of execution waves, the number of processes involved in the execution, the number of API calls observed from the last wave of each sample and the number of functions in the IAT of the unpacker's output. We match the results from the output of Minerva and PackerInspector with our ground truth data and Table 5.3 shows our results.

For the samples that execute in a single process, both Minerva and PackerInspector capture the number of processes and waves accurately. In two of these four samples, Minerva captures the five expected API calls accurately, and in the other two Minerva captures slightly more than the expected number. PackerInspector, however, attributes about 200x more API calls than the expected number to the final wave of the execution. Furthermore, Minerva builds the IAT for the two samples accurately and a slightly larger IAT for the other two. PackerInspector is unable to produce any output with an IAT, and there is no sign of API usage in the output of PackerInspector. The reason Minerva captures slightly more API calls than expected is that the compiler, naturally, adds various function calls around the source code. PackerInspector successfully identifies the correct number of execution waves but fails to attribute API calls accurately to unpacked code and also fails to produce any output with an IAT. Minerva, however, succeeds at both.

For the samples that perform multi-process execution, we observe that Minerva captures all processes, execution waves, API calls, and rebuilds PE files with the expected IAT. The reason Minerva does not capture slightly more API calls than the expected amount in these samples is that the final wave occurs within an injected process and does not contain the added functions from the compiler. Surprisingly, PackerInspector fails to detect multi-process execution in any of the samples, and we suspect this is because PackerInspector only monitors for multi-process execution via memory mapped files which none of the samples uses. From our multi-process samples, we observe the limitations of the original write-then-execute heuristic, in that it is unable to handle system-wide unpacking in a general and precise manner. However, the novel techniques introduced by Minerva are successful at this.

	Precision (Ground Truth, Minerva, PackerInspector)			
Sample	#Procs	#Waves	#API calls in final wave	#IAT size
(1) D1	1,1,1	2,2,2	5,8,1007	3,5,0
(1) D2	1,1,1	2,2,2	5,5,1007	3,3,0
(1) D3	1,1,1	3,3,3	5,5,1006	3,3,0
(1) D4	1,1,1	3,3,3	5,10,1007	3,6,0
(1) C1	2,2,1	2,2,1	6,6,†	3,3,†
(1) C2	2,2,1	2,2,1	6,6,†	3,3,†
(1) C4	2,2,1	2,2,1	1,1,†	1,1,†
(1) C5	2,2,1	2,2,1	5,5,†	3,3,†
(1) M1	2,2,1	3,3,1	6,6,†	3,3,†

Table 5.3: The evaluation results from matching Minerva and PackerInspector with the ground-truth samples of data set #1. † means not available because PackerInspector failed to reach the last wave.

When matched with our ground-truth samples it is clear that PackerInspector over-approximates the API usage of the applications, is unable to output unpacked code that shows API usage when faced with obfuscations of external dependencies and under-approximates the system-wide malware execution. These observations verify our hypothesis that state-of-the-art unpackers are unable to deal with many challenges faced by system-wide packing and that the techniques in Minerva overcome these limitations.

5.6.3 Empirical evaluation against selected malware

In our second experiment, we match Minerva and PackerInspector with the malware samples in data set #2. The goal of this experiment is twofold. First, we aim to measure how each unpacker captures system-wide unpacking based on the number of processes and waves they identify. Second, we aim to measure the difference in the total amount and the unique amount of API calls observed in each malware execution.

We only have access to PackerInspector via their web interface, and in this experiment, it is important to highlight the limitations of this. The samples we analyse with Minerva are executed in Windows 7, and the samples we analyse with PackerInspector presumably execute in Windows XP. In addition to this, we do not know the state of the execution environment that PackerInspector uses to execute the malware samples, such as the processes executing on the system, the network connection, the privilege-level of the malware, the security settings in the guest system, and so on. This adds a level of uncertainty to the results we present in this section since we are

not conducting an isolated comparison of techniques as the execution environments of the malware are, likely, significantly different. This is particularly relevant when dealing with the samples from data set #2 because malware samples are complex applications that are sensitive to their execution environment and small changes in the environment can have a substantial impact on the malware execution. However, we still feel it is appropriate to report the results as they give certain insights into the differences in our approaches. The results of our experiment are shown in Table 5.4.

In terms of multi-process monitoring, Minerva captures more injections than PackerInspector in eight samples and fewer injections in three samples. PackerInspector misses multi-process executions in all Tinba and Gapz samples. For the Tinba samples, PackerInspector only catches the first multi-process execution which occurs into the benign Windows process `winver.exe`, a process that is also started by each sample. PackerInspector misses all multi-process executions within the Gapz malware sample, and we believe there are two possible explanations for this. First, because PackerInspector exits prematurely, and second because Gapz uses the PowerLoader injection which does not rely on any of the API hooks used by PackerInspector to catch multi-process unpacking.

In one CryptoWall sample, Minerva finds one more multi-process propagation than PackerInspector, and in another sample, Minerva finds one less than PackerInspector. In both samples, both unpackers find two injections into `svchost.exe` and in both cases, PackerInspector also finds injections into `vssadmin.exe`. Minerva also finds an injection into `vssadmin.exe` in the sample with five process executions. However, we have found that these `vssadmin.exe` propagations correspond to false positives. In particular, we found no injections into `vssadmin.exe` but rather that the samples execute the following command `“vssadmin.exe Delete Shadow /All /Quiet”` using the `WinExec` call. We believe this call results in the memory flowing into the `vssadmin.exe` and is, effectively, the reason the unpackers identify execution in `vssadmin.exe`.

In two Ramnit samples, Minerva finds fewer injections than PackerInspector. In one of these samples, PackerInspector reports an additional injection into `IEXPLORE.exe` that is not identified by Minerva. We analysed the sample ourselves and found that the sample only injects into two `IEXPLORE.exe` processes if it fails to inject into two `svchost.exe` processes, which we observed by both Minerva and PackerInspector. However, researchers from Symantec [129] report that Ramnit also drops a file

Sample (family, md5)	Precision (Minerva, PackerInspector) Procs	Waves	API calls	Unique APIs
CryptoWall e73806e3f41f61e7c7a364625cd58f65	5(†2),4(†1)	8,4	7371,21050	148, 354
CryptoWall 5384f752e3a2b59fad9d0f143ce0215a	3, 4(†1)	6,4	9945,23580	135, 388
Tinba c141be7ef8a49c2e8bda5e4a856386ac	3,2	4,3	557,34076	54, 477
Tinba 08ab7f68c6b3a4a2a745cc244d41d213	3,2	4,3	667,49260	55,549
Tinba 6244604b4fe75b652c05a217ac90eeac	3,2	4,3	704,49262	55, 550
Gapz 089c5446291c9145ad8ac6c1cdfe4928	2,1	7,5	36509156, 15850000	140, 336
Gapz 0ed4a5e1b9b3e374f1f343250f527167	2,1	4,3	36504908, 15845670	125, 226
Gapz e5b9295e0b147501f47e2fcba93deb6c	3,1	5,2	36506063, 15844113	186, 251
Ramnit 448ce1c565c4378b310fa25b4ae3b17f	3, 4(†1)	8,5	6908,56720	116, 479
Ramnit 33cd65ebd943a41a3b65fa1ccfce067c	12(†1),5	30,6	16185,209828	153, 489
Ramnit 3bb86e6920614ed9ac5d8fbf480eb437	3, 5(†1)	8,8	3189, 115943	115, 621

Table 5.4: The evaluation results from matching Minerva and PackerInspector with the malware samples of data set #2. † indicates the number processes we determined to be false positives.

that will be loaded by each new instance of `IEXPLORE.exe`, which may be an explanation for why PackerInspector observes such an injection. However, we think it's most likely a result of over-approximation in PackerInspector as it would require the `IEXPLORE.exe` process to be launched on the system. In the other sample, PackerInspector finds an additional injection into a file with a random name which Minerva does not. Minerva, however, observes the creation of this file but does not see it execute. In the remaining Ramnit sample, Minerva captures seven more injections than PackerInspector, and both Minerva and PackerInspector identifies four injections into `IEXPLORE.exe` in this sample. Based on follow-up analysis, we determine six of the additional injections Minerva finds are true positives and one is a false positive. We conclude this because we found API-signatures that show injections into these processes, but not the remaining one.

In terms of precision for tracking API calls, there is a similar relationship between Minerva and PackerInspector as when matched with our ground-truth samples. In the samples from CryptoWall, Tinba and Ramnit, Minerva reports roughly twelve times fewer API calls within the malware execution than PackerInspector. We manually investigated several of the Tinba samples to confirm these numbers and found that Minerva captures API calls accurately within the malware execution. As such, we consider the API calls reported by PackerInspector to be a significant over-approximation. In addition to the total number of API calls, PackerInspector captures about three times more unique API calls in each malware execution, even in cases where Minerva

1	2	3	4-5	6-11
66%	15%	9%	5%	5%

Table 5.5: Number of process executions per malware sample.

1	2	3	4	5	5 <
52%	18%	7%	9%	2%	12%

Table 5.6: Number of waves per malware sample.

1	2	3	4	5	5 <
51%	17%	8%	3%	8%	13%

Table 5.7: Number of PE files constructed per malware sample.

1	2	3-5	5 <
56%	15%	18%	11%

Table 5.8: Number of Sections reconstructed per PE file.

0	1-10	11-15	16-20	21-25	26 <
18%	28%	7%	5%	2%	40%

Table 5.9: Number of imports per PE file.

finds more total API calls. The only instances where Minerva finds more total API calls are in the Gapz samples. The reason that there is a significant amount of API calls in these cases is that the Gapz malware scans a remote process for gadgets and this results in an enormous amount of calls to `ReadProcessMemory` (about 99.985% of calls in the Minerva analyses). We believe that the reason Minerva reports more API calls than PackerInspector only in the Gapz samples, is because PackerInspector exits prematurely.

When matched with these malware samples, it is clear that PackerInspector over-approximates the API usage of the applications, both in terms of total API calls and unique APIs used. We also find that in the majority of times, PackerInspector under-approximates the system-wide malware propagation and Minerva finds more system-wide unpacking.

5.6.4 Relevance on malware

In this experiment, we match Minerva with benchmark set #4. In total we run 119 samples through Minerva and collect (1) the number of process executions; (2) the number of waves; (3) the number of generated PE files; (4) the number of imports in the IAT of each PE file and (5) the number of sections in each PE file.

Table 5.5 shows the number of processes and Table 5.6 the number of waves in

our data set. We find that a third of the samples perform multi-process execution and that roughly half have multiple execution waves, which means that a large part of all the samples with single-process execution have multi-wave execution.

Table 5.7 shows the distribution of reconstructed PE files. We construct more PE files than the number of captured waves, which shows that some waves contain several regions that are non-related. Finally, Table 5.8 shows the number of sections reconstructed in each PE file and Table 5.9 shows the number of reconstructed imports. For roughly 20% of the PE files, we do not monitor any API calls in the code, and this is due to some PE files being a result of small amounts of taint in minor code regions.

5.6.5 Relevance on packers

In this experiment, we show that Minerva is relevant against publicly available packers from benchmark set #3. This experiment is common practice for unpacking engines [21, 112, 134, 147] and, therefore, natural for us to perform. We construct a simple application that will get the name of the current user and report back to us so we can verify the behaviour occurred correctly. We pack this application with 13 publicly known packers and analyse the samples in Minerva.

We show the results of our experiment in Table 5.10. The table shows the number of processes, waves, PE files, and whether we found the original code or a derivative thereof, and also whether we observed the original behaviour. Minerva produced PE files for most of the packers that are very similar to the original code, including correct API calls. In general, these packers are rather simple in comparison to some of the techniques we observe in malware from the wild. For example, all but one of the packers are single-process packers. This makes sense since the packers are not necessarily meant to be used by malicious software, but may be used by benign applications, which are not meant to inject into other applications. Furthermore, many of these packers rely on similar approaches for compression, e.g. the Lempel–Ziv–Markov chain algorithm, and the majority the packers used in these experiments are rather old.

5.6.6 Tinba case study

We now investigate in depth a case study of a real-world malware sample from the Tinba malware family[4]. Minerva outputs four PE files with sizes 12KB, 12KB, 16KB

[4]md5sum 08ab7f68c6b3a4a2a745cc244d41d213

Packer	#proc	#wave	#PE	U	OB
BoxedApp	1	1	1	Y	Y
Enigma	2	11	1	Y	Y
FSG_packed	1	2	2	Y	Y
mew11	1	3	4	Y	Y
MoleBox	1	4	9	Y	Y
mpress	1	2	2	Y	Y
PackMan	1	2	2	Y	Y
PECompact	1	4	4	Y	Y
PEtite	1	4	4	Y	Y
tElock	0	0	0	N	N
UPX	1	2	2	Y	Y
WinUpack	1	2	w	Y	Y
XComp	1	2	2	Y	Y

Table 5.10: The results from matching Minerva with known packers. **OB** indicates if we observed the original behaviour of the packed application. **U** indicates if we found the original code in Minerva's output.

and 24KB, respectively. We manually reverse engineered the sample to fully understand the system-wide propagation and where the sample exposes its unpacked code. The malware first decrypts memory from its data section and then transfers execution to this code. The decrypted code injects code into the Windows process `Winver.exe` and from `Winver.exe` it further injects into `explorer.exe`.

To inject code into `Winver.exe` Tinba launches a new instance of `Winver.exe` in a suspended state. Then, Tinba allocates memory on the heap of the newly started `Winver.exe` and copies some malicious code into this specific memory. Tinba then overwrites six bytes of the `Start` function in `Winver.exe` with the instructions `push ADDR; ret`, where `ADDR` is some address inside the dynamically generated malicious code. Effectively, Tinba ensures execution of its malicious code in `Winver.exe` by overwriting an initial function in `Winver.exe` to hijack execution.

Minerva captures one execution wave and outputs two unpacked PE files for the code in `Winver.exe`, one PE file based on a single execution wave in `explorer.exe` and also one PE file from the unpacked code in the initial process. The PE files produced by Minerva has 0, 11, 12 and 26 imports reconstructed. These results capture the execution perfectly because the PE file with 0 imports is purely the `push ADDR; ret` instructions of the malware execution trace in `Winver.exe` and the rest of the PE files contain various other stages with more payload content.

Minerva correctly identifies the malicious code, both the patched code of `Winver-.exe` and also the code on the heap that contains the core of the malware code.

More importantly, the PE files precisely capture the malware execution, and from the execution trace output by Minerva, we can precisely see the exact instructions `push ADDR; ret`. Minerva also catches the exact malware code inside of the `explorer.exe` process. The PE file captured from the second execution wave in the original malware process contains 11 imports in its reconstructed IAT, five of which are `ResumeThread, CreateProcessA, WriteProcessMemory, VirtualAllocEx` and `VirtualProtectEx`. A novice analysts can quickly determine that the execution performs a code-injection based on these API calls.

5.6.7 Performance evaluation

In the final part of our evaluation, we monitor the performance of Minerva. The reason we wait until now with performance evaluation is that most often both the work in Chapter 4 and the work of this chapter occur together and we, therefore, base our performance evaluation on experiments with both features enabled. The authors of PANDA report that recording gives a 1.85x slowdown in comparison to QEMU alone and replaying incurs a 3.57x slowdown [53]. This is expensive in comparison to systems that rely on hypervisor-based virtualisation for recording, e.g. AfterSight [38]. However, we consider PANDA's performance good enough for malware analysis in particular because the plugins that we deploy will have far more impact on the total analysis time. Naturally, the performance overhead in the recording stage can be used by the malware to evade analysis, and we discuss this further in Section 5.7. In this performance evaluation, we focus on the overhead of Minerva's analysis when replaying the recorded execution, and the numbers we report in this section are based on analysis of the 25 malware samples from data set #B introduced in Section 4.7.1.

The blue curve of Figure 5.5 shows the number of instructions replayed relative to the time taken in each of the analyses. On average the replay time is 3166 seconds, resulting in a 126x slowdown of the recording time, and we analysed on average 432365 instructions per second. In comparison, the developers of PANDA report a 24.7x slowdown when tainting data sent over the network and a 67.7x slowdown for tainting a 1KB file and encrypting it with AES-CBC-128 [53]. Additionally, Figure 5.6 shows the number of instructions that it took to replay the samples in our data set, and we observe that for about 90% of the samples this required less than 2 billion instructions.

Another interesting metric is the specific overhead incurred by Minerva-only code. Specifically, there is some share of the overhead that is due to the translation of QEMU TCG instructions to LLVM instructions and also overhead that is specific to

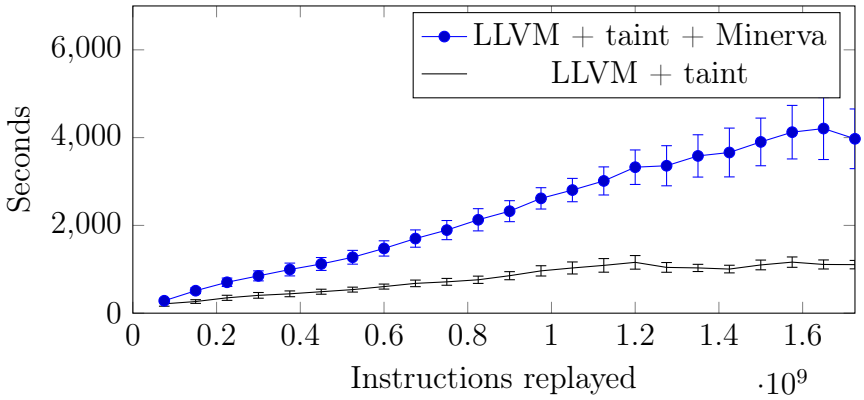


Figure 5.5: The average number and standard deviation of instructions replayed relative to time, for instruction counts where we have more than 3 samples executing the given number of instructions.

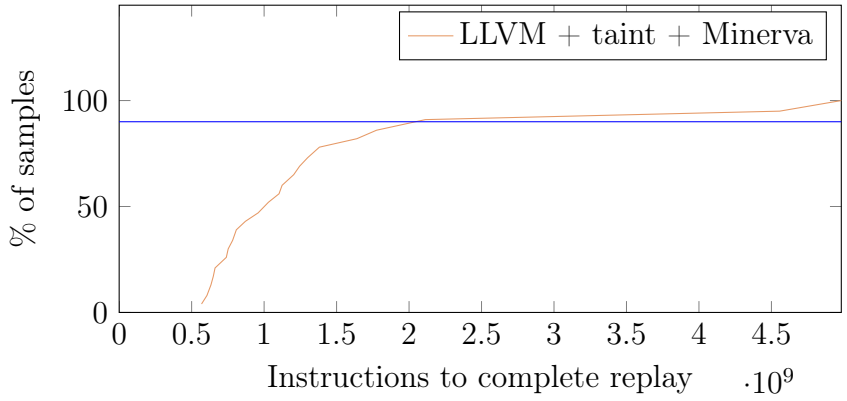


Figure 5.6: The amount of instructions needed to replay the samples in our data set. The horizontal blue line shows the 90% mark.

the taint implementation of PANDA. None of these requirements is strict to Minerva, in that we are not reliant on LLVM specifically, and PANDA's taint analysis does not focus on performance. Several systems focus on fast taint analysis [127, 22, 72] and, conceptually, the techniques of Minerva can be implemented on top of these taint libraries as well. To understand the overhead of Minerva's code, we ran the samples through a replay with LLVM-translation and taint analysis enabled, and no Minerva-specific analysis code. This gives us a reasonable estimate for how much of the analysis time was spent in the specific code related to Minerva. The black curve in Figure 5.5 shows these numbers. On average, each execution took 1275 seconds with the overhead of LLVM translation and PANDA's taint library. This corresponds to an average of 51x slowdown, meaning that Minerva's code takes up a bit more than half of the total 126x slowdown.

During the replay of a malware sample, Minerva does no checking to verify if the analysis is progressing, is stuck or something similar, and the numbers above report the total time of each analysis-replay. An interesting metric in addition to the total replay time is the time it took to reveal the instructions executed in each malware sample. In Figure 5.7 we show the time it took to uncover 95%, %99 and %100 of the unique instructions executed by the malware samples, respectively. The numbers decrease significantly, and on average it took 1614, 1964 and 2543 seconds to uncover 95%, %99 and %100 of the unique instructions executed, respectively. As such, it took roughly half of the total replay time to reveal 95% of the instructions in each sample.

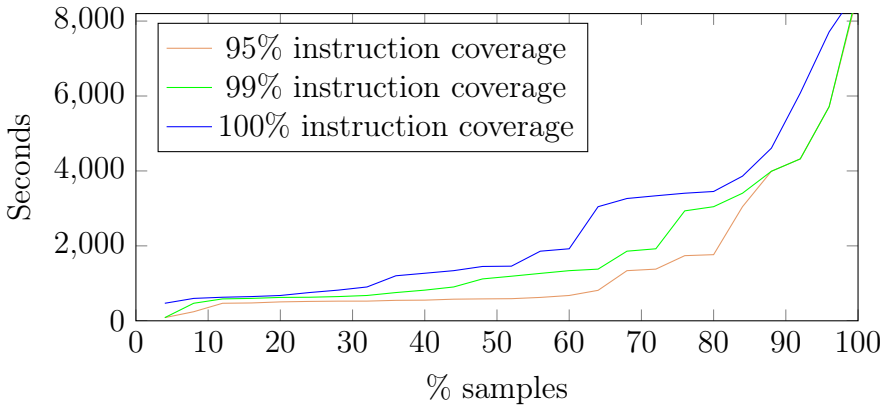


Figure 5.7: The time taken to explore the unique instructions in each malware sample.

The biggest performance bottleneck we found in Minerva is when malware makes the code execute longer via stalling loops. An example of a stalling loop from a Kovter sample[5] is shown in Figure 5.8. In total, the loop does 20 million iterations with sixteen calls to functions from the Windows API in each iteration. The loop has no real effect and is purely garbage code. In total, our 25-second recording of this sample reaches 17 million iterations before the recording is over and incurs a replay time of 3300 seconds.

The sample we observed with the longest replay time is from the Nymaim family that has a stalling loop with 1.4 billion iterations, and after the stalling loop, it calls the `Sleep` function from the Windows API to further stall the execution. In total, our 25-second recording of this sample took 170,000 seconds to replay.

[5]md5 of sample 147330a7ec2e27e2ed0fe0e921d45087

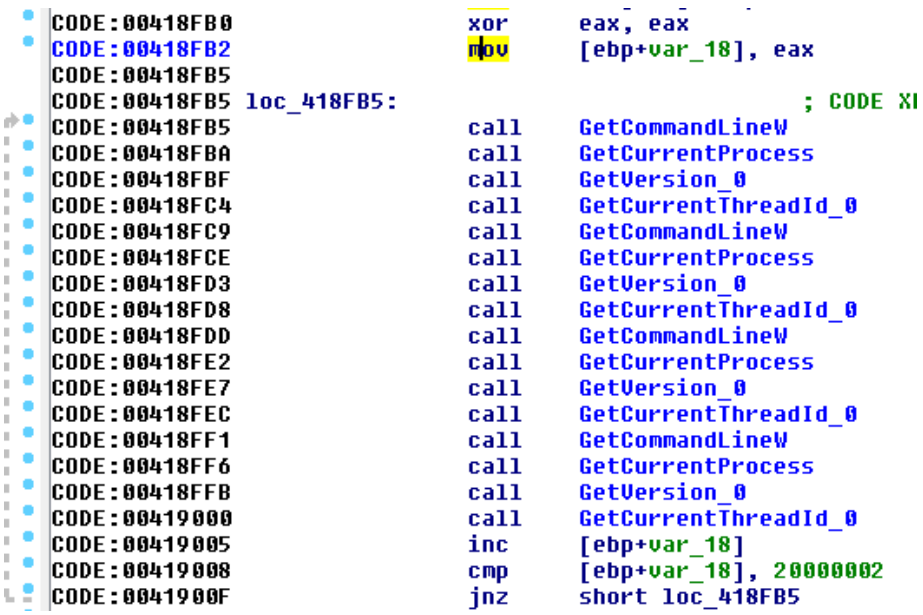


Figure 5.8: Stalling loop in Kovter malware.

5.7 Limitations

Stolen bytes and copying Windows API. Minerva's precise API capturing of API calls depends on monitoring whether the target address of branch instructions is the start of some Windows API function. Some malware use an anti-analysis technique, called *stolen bytes*, that copies a share of some API function to another place in memory to execute that code and then branch in the middle of the given API function. In this context, they avoid calling the beginning of the function and our technique will not capture it. One solution to this is to identify function boundaries for each API function and then monitor ranges rather than the function start.

In a more general setting, malware can copy entire functions or modules from the Windows API and then rely on the copied code rather than calling the original Windows code. In this context, Minerva will still capture whenever system calls happen, but new measurements should be taken for identifying the copying of Windows code. One approach is to mark library code with a specific taint label and then monitor whether library code is propagated. Naturally, this solution is subject to the limitations of taint analysis. Another approach is to incorporate forensic techniques that determine the similarity between the code in a given process and a set of external libraries, which we discuss further in the following paragraph.

Inlining and statically linked binaries. A limitation in Minerva in terms of

identifying external dependencies is when malware deploys inlined or statically linked code. The difference between this and copying external libraries as described above is that inlining and statically linking occurs at compile time where copying occurs at run time. Minerva is not capable of identifying inlined or statically linked external dependencies, and we consider this to be a slightly different problem, namely similarity analysis of the malware code with library implementations. However, inlining and statically linking can, of course, be used in combination with obfuscation techniques and similar, and, therefore, the problem becomes determining program equivalence in the general case, which is a well-known undecidable problem. Nonetheless, efforts can still yield positive and practical results as shown by previous work in areas such as library fingerprinting [58, 80], structural comparison of binary code [55, 63, 96] and, most recently, similarity detection via machine learning [105, 144].

Performance limitations. There are currently two main performance limitations in Minerva. First, malware can detect the presence of the recording component due to the 3.56x slowdown, and second, the replaying component limits the throughput of Minerva due to its performance cost. Stalling loops seem to pose a core limitation in this context. There is, however, previous work on how to deal with stalling loops in the context of full system emulation. Kolbitsch et al. implemented several features into the Anubis analysis system [90]. Their approach is to implement heuristics that detect when stalling loops occur and then either disable heavy instrumentation until the stalling loop exits or force execution out of the loop. The first approach is certainly possible to implement in Minerva, but it may run into issues if the stalling loop is also responsible for propagating executable malicious code since the taint analysis would likely be disabled. The second approach is more challenging to implement because replaying is not able to change execution-path in the guest system, as the execution is fixed to the replay log.

We consider five main avenues to improving performance. First, we can use hardware assisted virtualisation during recording and only full system emulation during replay, as suggested in AfterSight [38]. Second, we can implement various on-and-off analyses during the replay similar to Kolbitsch et al. Third, we can add light anti-analysis monitoring during the recording, for example, to limit the effectiveness of calls to functions like `Sleep`. In this case however, the implementation must use some form of approximation to determine the malware execution trace since taint analysis will not be available. Fourth, we can improve the speed of various parts in PANDA, such as the taint analysis plugin. Instead of converting instructions to

LLVM and performing taint analysis on the LLVM code, we can adopt the taint system by DECAF, which occurs directly on the QEMU tcg instructions [72]. Finally, an interesting avenue is implementing a feedback loop between record-and-replay that based on the analysis in the replay sends information to the recording about where a delay in execution occur and how to handle it. In this way, it is possible to incrementally build up a complete execution trace of the malware without anti-analysis tricks.

5.8 Related work

Automatic unpacking. We have already discussed several works on unpacking throughout the chapter [52, 75, 82, 83, 112, 140, 147]. Some of this work considers the concept of IAT destruction [82, 140]. We are, however, the first to propose an information flow model of dynamically generated code, and also a unified system for system-wide malware unpacking that handles obfuscation of external dependencies. The work by Ugarte et al. [147] highlights several missing gaps in existing unpackers and proposes a system-wide approach to unpacking. However, as we observed in this chapter, their approach is severely limited. In some aspects, Ugarte et al. provide a more refined model for dynamically generated code in that they assign various labels to the memory written by the malware based on whether it is executed and alike. These labels can easily be integrated into Minerva. In addition to this, they also highlight that several limitations in existing unpackers exist due to missing reference data sets, which indeed also motivated the construction of our synthetic benchmark set #1.

Malware disassembly. The work in this chapter is closely related to techniques that focus on disassembling malicious software. An accurate description of our work within this domain, rather than unpacking, is a system-wide malware disassembler. We gather a precise instruction-level execution trace of the malware and then gather more content to include in the reconstructed PE file with speculative disassembly. Traditionally, disassembly techniques are split between linear sweep, as used in GNU's Objdump, and recursive traversal [41, 141] algorithms. However, there are several pieces of previous work on disassembly that specifically target malware and these move beyond the traditional approaches. Kruegel et al. present an approach that combines a variety of techniques from control-flow analysis and statistical methods, in order to statically disassemble obfuscated binaries [97]. Kinder and Veith present an approach based on abstract interpretation that statically disassembles binaries and also resolves

indirect branch instructions [85]. Rosenblum et al. present a classification approach to identify function entry points [131], and Bao et al. [8] follow the same path and use machine learning and static analysis to identify functions within binaries. They train a weighted-prefix tree that recognises function starting points in a binary file and then value-set analysis [7] with an incremental control-flow recovery algorithm to identify function boundaries.

5.9 Chapter summary

In this chapter, we proposed extensions to Minerva that focus on generic and precise malware unpacking. From a technical point of view, Minerva deploys a concatic approach with both dynamic and static analysis and partitions the malware execution trace into execution waves based on information flow analysis. Minerva precisely monitors the API-calls of the malware code and accurately correlates these to the unpacked code. Based on the output of the dynamic analysis, Minerva performs static analysis on the execution waves to output a set of reconstructed PE files with valid import address tables and patched API calls.

From a theoretical point of view, Minerva deploys a precise model of execution waves based on an information flow model that captures dynamically generated malicious code *independently* of who wrote the code. We are the first to propose an information flow model for this problem. We came up with several novel algorithms that combine these execution waves with other artefacts collected from the dynamic analysis to carefully produce PE files that are well-suited for follow-up static analysis.

Finally, we proposed a new set of benchmark applications that exhibit unpacking behaviours with various forms of dynamically generated code, system-wide execution and import-address table destruction in order to address a missing gap in terms of ground-truth samples for testing unpackers. This benchmark suite is the first of its kind in that previous benchmark data sets for testing automatic unpackers rely on third-party applications to perform the packing.

We evaluated Minerva against our synthetic applications, real-world malware samples and also performed a comparative evaluation. Our results show that Minerva is significantly more precise than previous work and outputs unpacked code that shows external dependencies, which previous work does not. Our results also show that Minerva captures system-wide unpacking in many cases where previous work fails.

Chapter 6

A characterisation of system-wide propagation in the malware landscape

> "*A very great deal more truth can become known than can be proven.*"
>
> — Richard Feynman, The Development of the Space-Time View of Quantum Electrodynamics, Nobel lecture, 1965.

System-wide propagation is frequently observed in malware, and there are several resources, like blog posts and similar, that detail some of the techniques used. However, there is currently no thorough study on the subject at large, and the full extent of system-wide malware propagation remains unknown. In this chapter, we perform a systematic study on many real-world samples to comprehensively characterise system-wide propagation within the malware landscape. The goal is to use detailed and precise analyses to derive high-level views. To achieve this, we collect a diverse set of malware samples, analyse them in Minerva and then extract vast amounts of statistics about the results. We use these results to provide an in-depth discussion centred on four main research questions.

6.1 Introduction

In the previous chapters, we have introduced several new problems in malware analysis and proposed novel solutions. The problems that we presented were all centred around system-wide malware propagation, and we have spent significant effort in the details of this subject. However, we have so far postponed a thorough assessment of system-wide propagation's full extent in the wild, and we set out to do that in this chapter.

Systematic studies of large malware data sets are essential for anti-malware research. We rely on these studies to systematise our knowledge, design new experiments that verify our techniques and to explore new research avenues. Although system-wide malware propagation has received the attention of malware authors for several years, the academic community has given it limited attention, and there are currently no comprehensive studies on the subject. Most resources are anecdotal examples of code injection techniques [71, 120] and blog posts by anti-malware researchers [16, 74], and these mainly present results based on manual analysis of specific samples.

In this chapter, we aim to fill this gap by performing the first large-scale systematic study of system-wide malware propagation. To this end, we build a comprehensive statistics component around Minerva and then use our framework to analyse and characterise a large and diverse set of malware samples. Our study is focused on three aspects of system-wide propagation. First, to understand the prevalence and diversity of system-wide malware propagation. Second, to understand the relationship between system-wide malware propagation and malicious activities. Third, to understand system-wide malware propagation evolution over time and inter-family characteristics of malware propagation strategy.

To reason about system-wide malware executions at large, we define the concept of a system-wide control-flow graph, which is a data structure that describes the entire system-wide malware execution. Effectively, this data structure combines the ability to identify multi-process execution from Chapter 4 and capture execution waves from Chapter 5 to give a complete overview of a given malware propagation.

We perform our study with a comprehensive set of malware samples that reflects a diverse and broad view of malware in the wild. The data set we collect has samples from many different families, many different malware types and samples that were discovered over a seven-year time-frame. In addition to this, our data set is completely balanced in terms of samples per family to avoid biases in the results.

In summary, this chapter makes the following contributions:

- We propose the concept of a system-wide control flow graph which captures intrinsic aspects of malware propagation in a given sample.
- We collect a data set with a broad range of malware samples that were discovered across the last eight years.
- We implement a comprehensive and rigorous statistics component that interprets the results of Minerva.

- We characterise system-wide malware propagation through a detailed and thorough study of the malware samples by analysing them in Minerva and gathering many different and insightful statistics.

6.2 System-wide propagation graph

In this section, we introduce the concept of a system-wide propagation graph (SPG). Intuitively, SPG combines the output of the techniques from Chapter 4 and Chapter 5 into one single data structure. This data structure, thus, captures intrinsic aspects of host-based malware propagation in terms of multi-process execution and dynamically generated code. Informally, the SPG is a directed graph where the nodes constitute execution waves, and the edges constitute transitions from one execution wave to another. Each node then contains meta information to capture useful properties, such as their particular process. Formally, the SPG is described as follows.

Definition 4. *A system-wide propagation graph is a 3-tuple, SPG=(V, E, $\hat{v}$), where G=(V, E) is a directed weakly-connected graph such that each node is an execution wave, each edge is a control-flow transition, and $\hat{v} \in V$ is the entry point.*

To capture the SPG in practice, we must be able to identify all of the execution waves and the corresponding control-flow transitions for a given malware sample. We already know how to capture execution waves from Chapter 5 and we know how to identify control-flow transitions between execution waves across processes from Chapter 4. To capture control-flow transitions from waves internally within a process, we make an edge from the last instruction executed before the entry of a new execution wave.

To reason quantitatively about malware propagation we build several definitions on the SPG and the general idea is to describe malware propagation techniques in more ways than only counting the number of processes and the number of waves. We give insights about the malware propagation topology by creating two definitions that capture the maximum depth of processes and waves and one definition that describes the width of the SPG.

Definition 5. *Given an SPG, G=(V, E, $\hat{v}$), the process-depth of PD(G) is the maximum number of different processes visited amongst all non-cyclic path through G, starting from $\hat{v}$.*

Definition 6. *Given an SPG, G=(V, E, $\hat{v}$), the wave-depth WD(G) is the size of the longest non-cyclic path through G, starting from $\hat{v}$.*

Definition 7. *Given an SPG, G=(V, E, $\hat{v}$), the SPG-width |G| is the maximum number of non-cyclic paths from the entry-point $\hat{v}$ to all of the leaf nodes in the graph.*

Notice that Definition 5 and Definition 6 are different from calculating the total number of processes and waves, respectively. The notion of depth in the malware execution is relevant because it encapsulates how deep into the malware execution a manual malware analyst must go before reaching the end of an injection/unpacking sequence, for example, in a debugger. The notion of SPG-width given in Definition 7 aims to capture how broadly malware executes on a system, and, therefore, describes the number of execution paths in the malware propagation rather than how deep they are.

6.2.1 Example of system-wide propagation graph

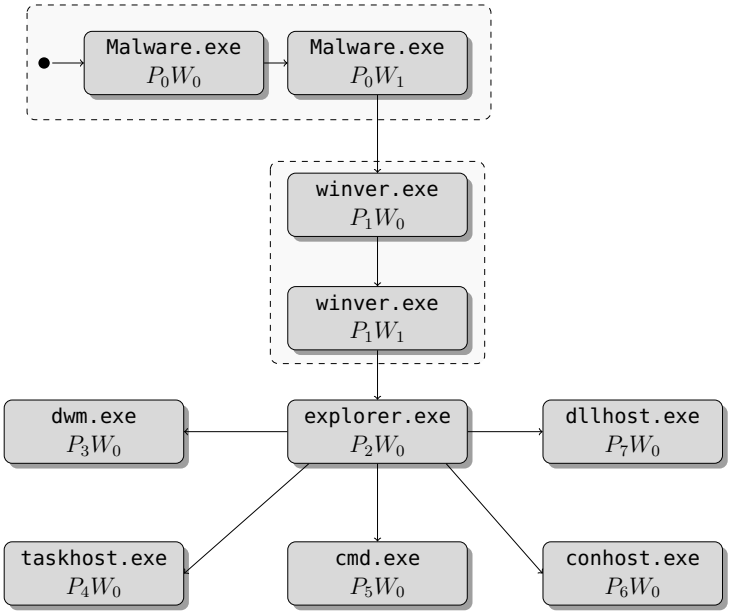
To illustrate system-wide propagation graphs, as well as process-depth, wave-depth and SPG-width, we show the SPG of a malware sample from the Tinba family in Figure 6.1. The sample initially creates one wave of dynamically generated code in its initial process and then launches a new process of the otherwise benign Windows application `winver.exe`. The malware injects code into the launched `winver.exe` process and the code in `winver.exe` further creates another wave of dynamically generated code. The second wave in `winver.exe` injects code into the otherwise benign Windows process `explorer.exe` and inside this process, the malware proceeds to inject code into five other processes on the system. The process-depth of the SPG is four, the wave-depth is six, and the SPG-width is five.

6.3 Research methodology

We now elaborate on the methodology for our large-scale empirical study. Specifically, we outline our data set, our research questions and also describe our experimental setup, including false-positive removal. A checklist for prudent evaluation is given in Table B.3 in Appendix B.

6.3.1 Data collection

The goal of our data collection is to establish a set of samples that broadly represents the Windows malware landscape over several years. To this end, we have collected samples from various kinds of malware, a diverse set of families and from an extended



$$
\begin{aligned}
G &= (V, E, \hat{v}) \\
V &= \{P_0W_0, P_0W_1, P_1W_0, P_1W_1, P_1W_2\} \\
E &= \{\{P_0W_0, P_0W_1\}, \{P_0W_1, P_1W_0\}, \{P_1W_0, P_1W_1\}, \{P_1W_1, P_2W_0\}, \\
&\quad \{P_2W_0, P_3W_0\}, \{P_2W_0, P_4W_0\}, \{P_2W_0, P_5W_0\}, \{P_2W_0, P_6W_0\}, \\
&\quad \{P_2W_0, P_7W_0\}\} \\
\hat{v} &= P_0W_0 \\
PD(G) &= 4 \\
WD(G) &= 6 \\
|G| &= 5
\end{aligned}
$$

Figure 6.1: The system-wide propagation graph of Tinba malware sample.

timeline. We identify the year a sample is from as the year the sample was first submitted to VirusTotal. In contrast to other quantitative large-scale studies [14, 54, 147] that analyse tens and hundreds of thousands of malware samples, we instead keep the total samples in our data set comparatively low and comprehensible but with a wide distribution of malware families. This is because samples within the same family tend to behave similarly [121], and by keeping the numbers low, we enable more investigations to understand and verify results at a detailed and complete level.

Most of the samples in our data set are collected from `kernelmode.info`, `VirusTotal`, and a few are from `virusshare.com`, `contagiodump.com` and Malpedia [121]. To ensure that we had no benign samples we required each sample to be detected by at least 13 anti-malware vendors and to ensure consistency of the samples belonging to a given malware family we required for each sample that at least two vendors label it within the same malware family. On average each sample had 51 anti-malware vendors report it as malicious and a median of 53.

Our data set contains 65 different malware families with ten samples from each family, making a total of 650 samples. Table 6.1 shows the specific families and the years their samples were first discovered, and Figure 6.2 shows the yearly-distribution of when samples were first discovered. Our data set has samples spanning 2012-2018 and Figure 6.3 shows the number of families represented each year. We have an average of 22 families represented each year, with a maximum of 33 families in 2014 and a minimum of 19 families in 2012.

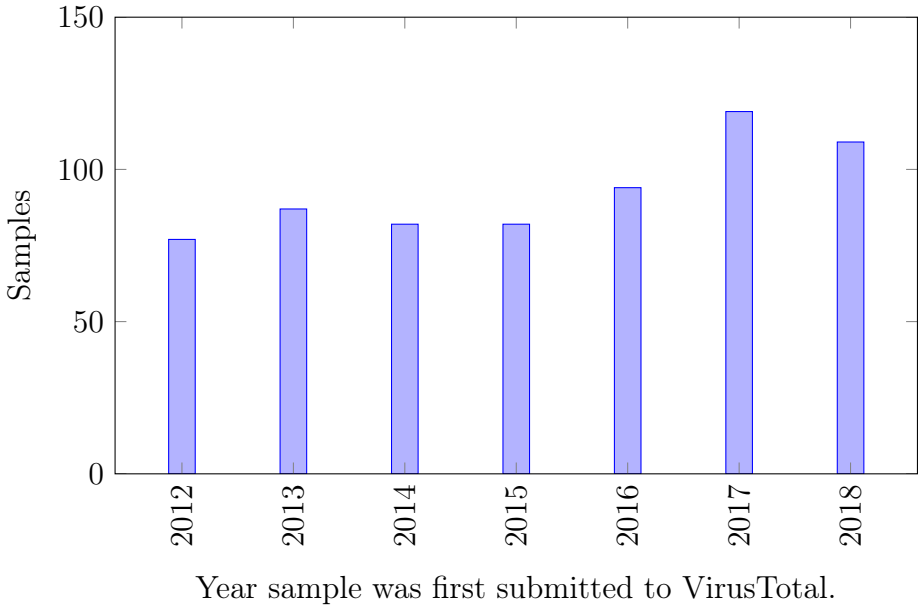


Figure 6.2: Yearly-distribution of samples in our data set.

Family	Discovered	#samples	Family	Discovered	#samples
Androm	2012-2014	10	Midie	2013, 2016-2017	10
Artemis	2012	10	MiniDuke	2013, 2017	10
Barys	2012-2013, 2015	10	Mira	2014, 2016-2017	10
Bitman	2015-2016	10	Natas	2017-2018	10
Buzus	2013, 2016, 2018	10	Neshta	2012-2014	10
CTBLocker	2015-2016	10	Neshuta	2012-2014	10
Cerber	2015-2017	10	Ngrbot	2012, 2014-2018	10
CoinMiner	2013-2015, 2017	10	Nimnul	2013-2017	10
CosmicDuke	2013-2014	10	Nitol	2013, 2015	10
Crowti	2014-2016	10	Nymaim	2013, 2015-2018	10
Cryptlock	2013-2014, 2016	10	Otwycal	2017-2018	10
Cutwail	2014-2016, 2018	10	Padodor	2017-2018	10
DealPly	2013-2014	10	Parite	2013, 2015, 2017	10
Dorkbot	2012, 2013, 2015, 2018	10	Pony	2014, 2018	10
Dridex	2014-2016	10	Pronny	2018	10
Eldorado	2013-2014	10	Ramnit	2012, 2014-2017	10
Emotet	2012, 2014-2017	10	Razy	2013-2014	10
Fareit	2013, 2015-2017	10	Renos	2014, 2018	10
Flood	2013, 2015-2017	10	Rovnix	2014, 2016	10
Fujacks	2014-2015, 2018	10	Sality	2014, 2016-2017	10
Fynloski	2012-2013, 2015-2016, 2018	10	Shifu	2015, 2017	10
Gamarue	2012, 2017	10	Simda	2012, 2014-2015	10
Gootkit	2014-2015	10	Symmi	2012-2014	10
Kasidet	2015, 2018	10	TeslaCrypt	2015-2016	10
Kazy	2013-2014, 2017	10	TinyBanker	2012, 2017	10
Kovter	2013-2016	10	Urausy	2012	10
Kraddare	2012-2014	10	Ursnif	2015-2016, 2018	10
Kryptik	2012-2015, 2017	10	VBKrypt	2013-2014, 2016-2017	10
Madangel	2013, 2015, 2017	10	Vawtrak	2013, 2015-2016, 2018	10
Madi	2012	10	Wannacry	2017-2018	10
Mamba	2018	10	Waski	2013-2015, 2017	10
Mazam	2016	10	Zbot	2012-2014	10
Vilsel	2014, 2016-2017	10			
Total families: 65 **Total samples: 650**					

Table 6.1: The malware families that we used for our study.

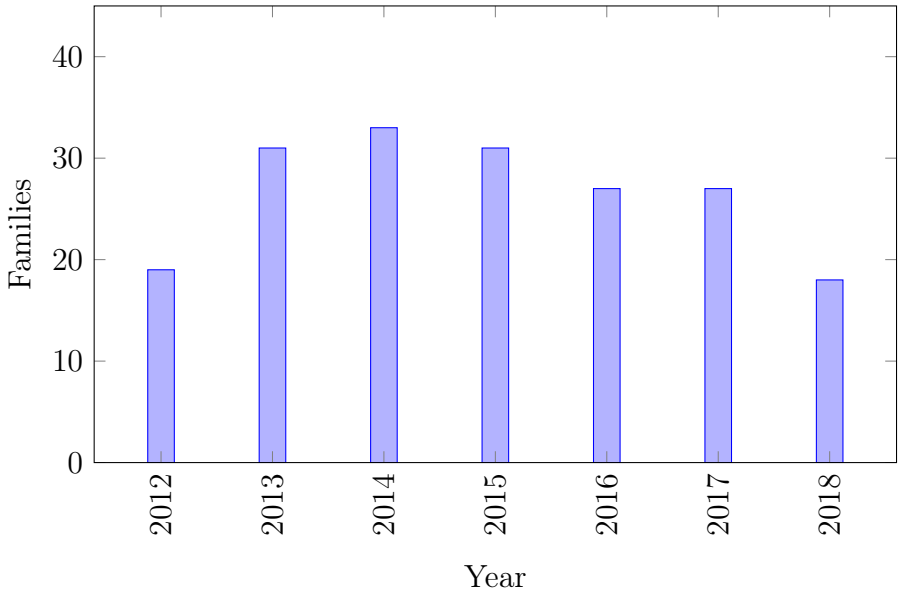


Figure 6.3: Yearly-distribution of families.

6.3.2 Research questions

The goal of our study is to characterise malware propagation at large and to do this, we propose four high level research questions.

1. Is system-wide malware propagation prevalent?

   The goal of this research question is to give quantitative measurements of the basic characteristics of malware propagation at large. To answer this, we gather statistics on the number of processes and execution waves in each malware sample and also the depths and width of the SPG.

2. Are the propagation strategies used in the wild diverse?

   The goal of this research question is to give insights about the individual propagation strategies used by the malware samples. To answer this, we investigate (1) which processes are the targets of malware propagation, (2) which specific techniques are used for malware propagation and (3) the proportion of droppers versus code injections.

3. Are there clear relations between malicious behaviour and system-wide malware propagation?

   The goal of this research question is to identify how malware performs its malicious behaviours, e.g. achieves persistence, escalates process-level privileges or

connects back to its C&C server, relative to its system-wide propagation strategy. To answer this question, we investigate where in the SPG malware: (1) uses dynamically generated code, (2) executes most of its code and (3) performs sensitive API calls.

4. Has system-wide malware executions changed consistently towards one direction and are there any clear inter-family SPG consistency?

   The goal of this research question is to measure if malware propagation has changed over time and if samples from the same family propagate similarly. To answer this question, we gather statistics based on grouping our data set into the years the samples were first observed and also do family-oriented statistics.

6.3.3 Experimental set up

We conduct all of our experiments on a 4-core Intel-7 CPU with 4.2 GHz and a Windows 7, 32-bit guest architecture. The guest is in a closed network and connected to another virtual machine that performs network simulation using InetSim[76]. As such, malware samples that connect back to some C&C server will be able to resolve DNS names, connect to every IP and also receive content. However, the content itself is the default data provided by INetSim, which will hinder malware samples from getting C&C-specific data, e.g. bot commands. We set a recording time of 25 seconds and a loft of 120 minutes for replaying. Due to the resources required to run this type of experiment, we analysed each sample once.

The applications on the guest machine were executed with a local admin account and with User Account Control (UAC) enabled. We perform no user stimulation during the analysis, and no applications were running in the guest system apart from the generic Windows processes. To start execution of a given sample, we mount it as an `.iso` file in QEMU and then send keystrokes to QEMU to execute command-line commands in the guest's terminal for opening the mounted `.iso` and starting the sample.

6.3.4 Generating multi-process signatures

In Chapter 4 we proposed the concept of malware execution trace and techniques to identify intrinsic aspects of process-to-process control flow transitions. However, the goal of the study in this chapter is to give a precise view of system-wide propagation and we need to ensure we distinguish malware samples that use different multi-process propagation techniques, while simultaneously identifying samples that

rely on the same technique. To this end, we use the output of Minerva to highlight multi-process propagation and then manually refine this information into easily recognisable signatures. We use the precise API capture of Minerva, the system-wide malware execution trace and identification of intrinsic aspects in multi-process transitions to speed up the reverse engineering process significantly. We started our study with no signatures and then incrementally created signatures for each analysed sample.

6.3.5 False positive and false negative elimination

In Section 4.7.3.2, we showed that in some cases, Minerva may over-taint the system and include benign code in the malware execution trace. To elegantly solve this issue, we introduce a heuristic that identifies over-tainting in order to eliminate false positives.

To identify over-tainting, we first check for each sample with multi-process execution if any of our generated signatures recognise the multi-process transitions. If there are some processes with malicious execution but no matching signature, then we consider this process a potential case of over-tainting. We determine this by matching the tainted code with code that was present in the system before the malware execution. The idea is that if the code execution is due to over-tainting, it must be part of the non-malware code and, therefore, part of the original code as it was before the malware execution. We verify this in practice by matching API calls observed by Minerva and the API calls performed by the potentially benign code. If we find more than 99.00% of the API calls from the tainted code correspond to API calls in the existing Windows code or if the tainted code has no API calls in the process at all then we declare the process a false-positive. Otherwise, we consider it a true positive and reverse engineer the code in order to create a signature. The reason we maintain a 99% threshold is to allow for incompleteness in our ability to extract API calls statically from the Windows code and Minerva does this as part of our static analysis post dynamic analysis. So far we have not found any false positives from this approach and in reference to our false positive analysis in Section 4.7.3.2, specifically in Table 4.5, running the same experiment with our false positive analysis removes all four of the false positives.

Finally, PANDA itself may fail in some instances and only replay little of the recorded execution. To deal with samples that did not execute properly, we remove all samples with less than 25 different instructions executed on the system.

All of the statistics we report in this chapter are post false-positive elimination, including the input data set described in 6.3.1. As such, we have manually verified multi-process execution for all samples in our data set as a result of our signature creation. The numbers we report are, therefore, a strict lower-bound, but we may miss out on some aspects of the malware capabilities due to the nature of dynamic analysis. Although a strict lower-bound, we consider it a precise estimate, and often significantly more accurate in comparison to other work, as justified by the previous chapters.

6.4 Experimental results

In this section, we present the results of our study in terms of the four research questions described above. This section focuses on a quantitative presentation and then in Section 6.5, we give an interpretation and qualitative discussion of the results.

6.4.1 Quantitative measurements

We now present results related to our first research question "*Is system-wide malware propagation prevalent?*" by gathering statistics about the core attributes of the SPGs in our entire data set.

The first statistic we gather is the number of processes involved in each malware execution, and Figure 6.5 shows the results. In total, 151 of the 650 samples exhibited multi-process propagation and 67 of these samples split into two processes. This means 23.23% of all samples use multi-process propagation and 55.6% of malware that performs multi-process propagation do so in three or more processes. The 151 samples are from 40 different families corresponding to 62% of all families, and Table 6.2 shows the complete list of families and their respective number of multi-process samples. Madangel is the only family where all samples use multi-process propagation, and we observed four families where all but one sample deployed multi-process execution, namely Emotet, Razy, Natas and TinyBanker. The maximum number of processes we observed in a single malware execution is eleven, which occurred in three samples from the Natas family and Figure 6.4 shows the SPG of one of these samples[1]. The malware initially transitions through three processes via files that it dropped on the system and then at a process-depth of four injects code into every process on the system for which it has permissions.

[1] `SHA-256 sum bd037ee28fc97f59525f384136fc067dded691230597384f04b10efe360055d1`

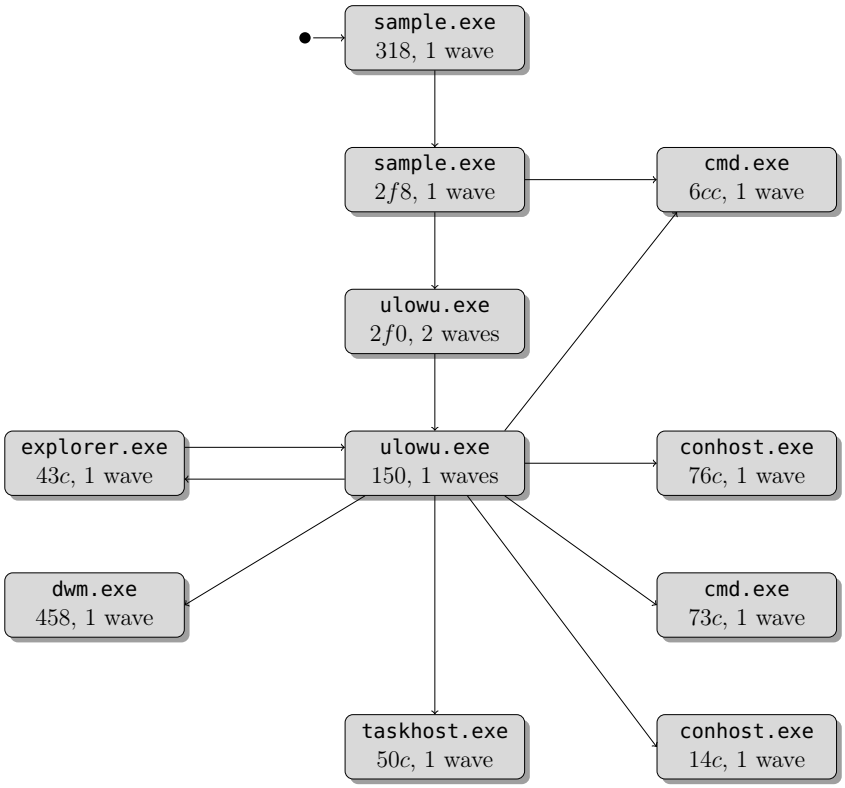


Figure 6.4: The system-wide propagation graph of Natas sample. Only processes are shown, and each process box contains the `PID` and the number of waves for the given process.

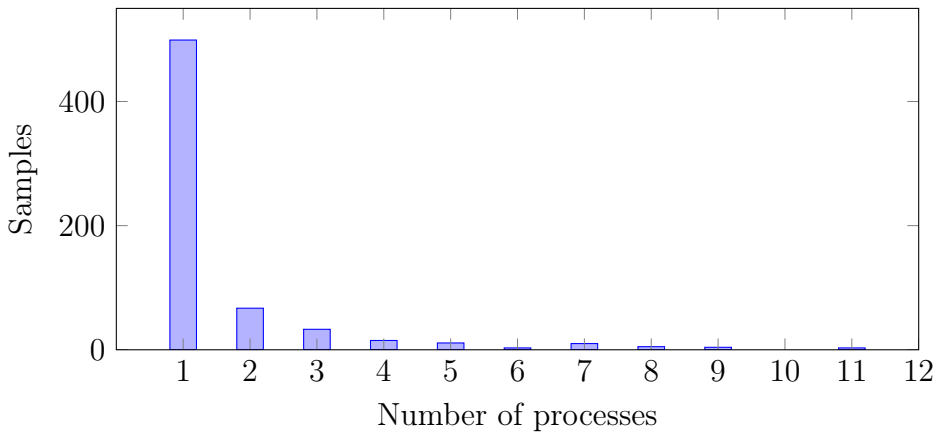


Figure 6.5: Number of processes involved in malware executions of all samples.

Family	$\|\mathcal{M}\|$	Family	$\|\mathcal{M}\|$	Family	$\|\mathcal{M}\|$	Family	$\|\mathcal{M}\|$
Madangel	10	CoinMiner	5	Ursnif	3	Fynloski	2
Natas	9	Nitol	5	Dridex	3	Cerber	1
Emotet	9	Midie	4	CTBLocker	3	Barys	1
TinyBanker	9	Urausy	4	Kryptik	3	Bitman	1
Razy	9	Vawtrak	4	VBKrypt	2	Fareit	1
Gamarue	7	Dorkbot	3	Androm	2	Waski	1
Ramnit	7	Zbot	3	Buzus	2	TeslaCrypt	1
Mira	7	Kasidet	3	Cutwail	2	Eldorado	1
Sality	6	Kovter	3	Crowti	2	Shifu	1
Nimnul	6	Symmi	3	Nymaim	2	Ngrbot	1

Table 6.2: Families with multi-process propagation. $|\mathcal{M}|$ denotes the number of multi-process samples in the given family.

Process-depth	1	2	3	4	5
Samples	499	97	23	14	17

Table 6.3: Process-depths of all samples.

The process-depths exhibited by the samples are shown in Table 6.3 and, in total, 91.7% of the samples have a process-depth of either one or two. This is more samples than the number of samples with one or two process executions (87%) confirming that some malware split execution rather than incrementally build a chain of multi-process executions. We found a maximum process-depth of five, and this occurred in seventeen samples. The complete list of process-depths for all 40 families with multi-process propagation is shown in Table 6.4. We find that 17 families have multiple samples with multi-process execution that do not exhibit the same process-depth, and, on the other hand, we find that in 12 families with multiple multi-process samples all samples with multi-process execution within their respective family exhibit the same process-depth.

The SPG-widths of all the samples are shown in Table 6.5. In total, 597 of the samples have an SPG-width of one, which means the vast majority of samples sequentially propagate through processes. The maximum SPG-width we found was seven, and this occurred in five malware samples, three of which belong to the Natas family, one from the Shifu family and one from the Ramnit family. We found a total of 20 families with at least one sample that has SPG-width larger than one, and the full list of these families are shown in Table 6.6 alongside the number of samples in each family that deploys a given SPG-width.

The distribution of execution waves is shown in Table 6.7. In total, 393 of 650 samples have multiple execution waves, which correspond to 60% of the samples, and we found thirteen samples that deploy more than 25 execution waves. The three

Process-depth	1	2	3	4	5		1	2	3	4	5
Androm	8	1	1			Midie	6	3			1
Barys	9		1			Mira	3	7			
Bitman	9	1				Natas	1	4			5
Buzus	8			1	1	Ngrbot	9	1			
CTBLocker	7	3				Nimnul	4	3	1	2	
Cerber	9	1				Nitol	5	5			
CoinMiner	5	5				Nymaim	8	2			
Crowti	8	1	1			Ramnit	3	3	2	1	1
Cutwail	8	2				Razy	1	9			
Dorkbot	7	1	2			Sality	4	6			
Dridex	7	2	1			Shifu	9		1		
Eldorado	9	1				Symmi	7	1	1	1	
Emotet	1	1	1	5	2	TeslaCrypt	9	1			
Fareit	9	1				TinyBanker	1		7	1	1
Fynloski	8	2				Urausy	6	1			3
Gamarue	3	7				Ursnif	7	1		2	
Kasidet	7	3				VBKrypt	8				2
Kovter	7	1	1	1		Vawtrak	6	2	2		
Kryptik	7	3				Waski	9	1			
Madangel		10				Zbot	7	1	1		1

Table 6.4: Families with at least one sample that has process-depth higher than one. The columns indicate a given process-depth and the rows display the number of samples in a given family with the given depth.

SPG-width	1	2	4	4	5	6	7
Samples	597	24	4	6	9	5	5

Table 6.5: SPG-width of all samples.

SPG-widths	1	2	3	4	5	6	7
Androm	9	1					
CTBLocker	7	3					
Cerber	9	1					
CoinMiner	9		1				
Cutwail	8	2					
Dorkbot	8				2		
Emotet	3	4		2	1		
Kovter	9	1					
Madangel	8	2					
Midie	9	1					
Natas	5				2		3
Nitol	5	5					
Nymaim	8	2					
Ramnit	5	1	3				1
Razy	9	1					
Sality	4				1	5	
Shifu	9						1
TinyBanker	8			2			
VBKrypt	8			2			
Vawtrak	7				3		

Table 6.6: Families with at least one sample that has SPG-width higher than one. The columns indicate a given SPG-width and the rows display the number of samples in a given family that deploys the given SPG-width.

Execution waves	1	2	3	4	5	6	7	8	9	10	11-15	16-25	> 25
Samples	257	123	97	41	17	31	7	12	6	7	26	13	13

Table 6.7: The number of execution waves of all samples.

highest counts of execution waves are 54, 354 and 5389, which came from samples in the Pronny, Artemis and Urausy malware families, respectively. In figure 6.6 we show the average number and standard deviation of execution waves per family in all families except Pronny, Artemis and Urausy, who have an average of 43, 37 and 545 execution waves, respectively, and, a standard deviation of 2, 52 and 807. We observe seven families that have no execution waves, meaning they do not deploy any dynamically generated code, and the average number of execution waves is slightly less than 15 if we discard the three families mentioned above. Additionally, the standard deviation is reasonably consistent across the majority of families, however, certain families show a considerable variation and the following families: Urausy, Artemis, Shifu, Buzus, Mira, Androm, Ursnif, Symmi and VBkrypt each have a standard deviation that is larger than their respective average number of execution waves.

6.4.2 Propagation intrinsics

We now present results related to our second research question "*Are the propagation strategies used in the wild diverse?*". To do this, we identify the processes that are targets of multi-process propagation, identify the API routines used by malware to propagate through the system, discuss detailed case studies of interesting malware samples, and also measure the proportion of code injections versus propagations via dropped files.

6.4.2.1 Target processes

Table 6.7 shows the names of the most targeted processes and the number of times they were used for multi-process propagation. The most popular target is malware starting another instance of itself, in this case `sample.exe`. The second most popular target is `explorer.exe` and following this are `iexplore.exe`, `cmd.exe` and `conhost.exe`, which are all part of the Windows OS. In fact, the following eight Windows applications `explorer.exe`, `iexplore.exe`, `cmd.exe`, `conhost.exe`, `taskhost.exe`, `dwm.exe` `svchost.exe`, and `winver.exe` make up 241 targets out of 394 multi-process propagations, corresponding to roughly 61%. Furthermore, if we discard the multi-process executions via `sample.exe` these eight applications make up 75% of the victims.

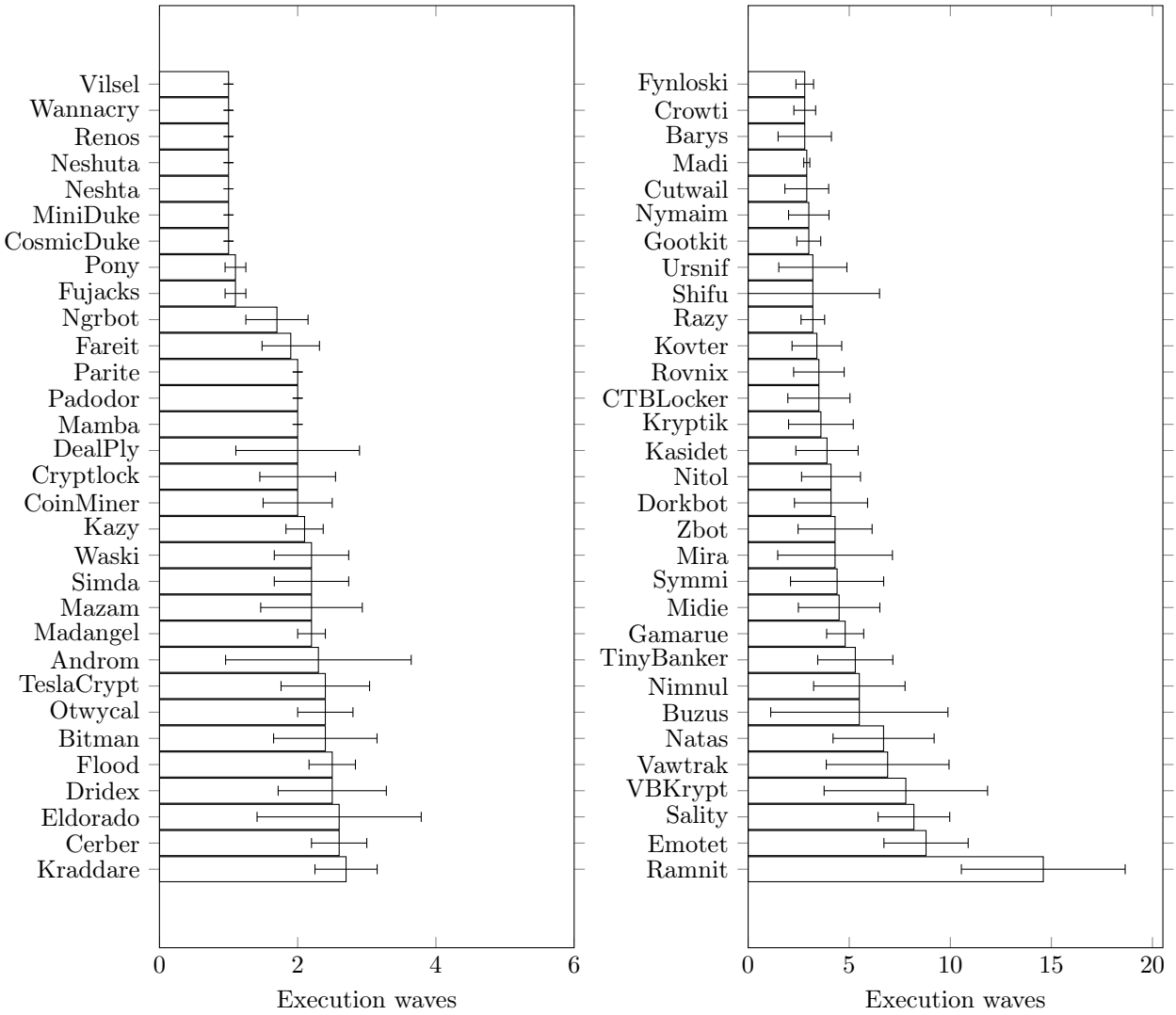


Figure 6.6: Average number and standard deviation of execution waves per family. The plots are sorted by average starting from the top left, i.e. notice the difference in x-axis values between the two plots.

In total, we found 244 multi-process executions targeted processes with names from the Windows OS, and 150 multi-process executions with proprietary names where `sample.exe` takes up 74 of these. As such, there are only 76 cases where the malware execute via programs with non-standard names, corresponding to 19% of the total multi-process propagations.

In Table 6.8, we show the per-family process targets as well as the number of samples in each family that execute in the given target. The table also shows the size of the group of samples that share most target processes and the size of this group over the total samples in the family. Finally, the table shows the number of

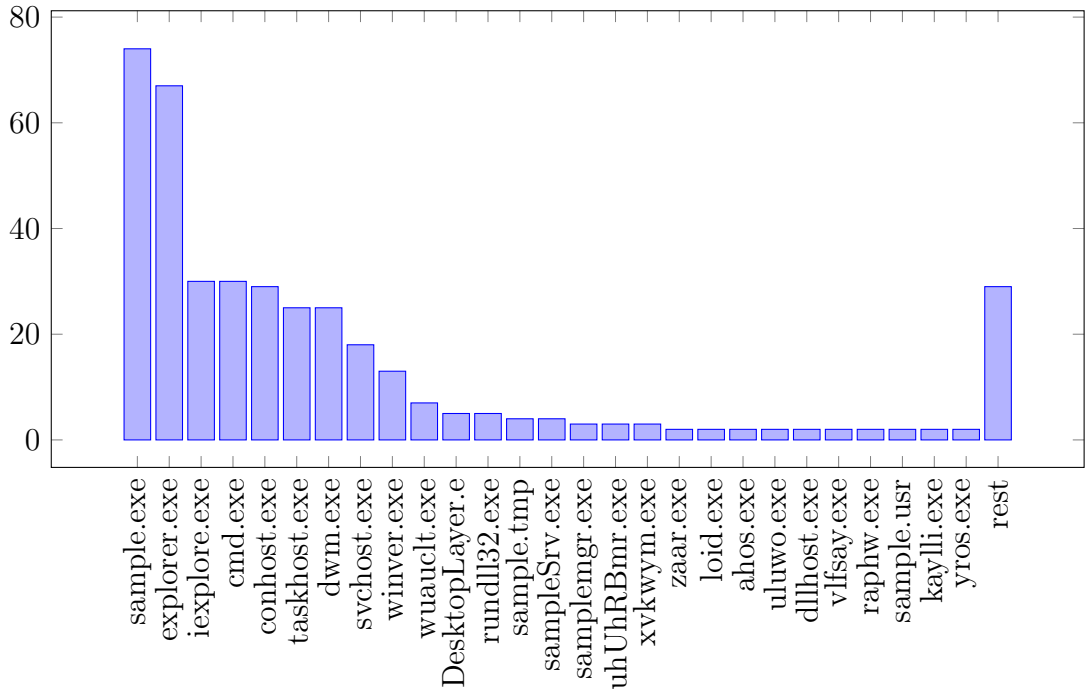


Figure 6.7: The processes that are the most popular targets for multi-process propagation.

samples in the group just mentioned multiplied by the number of targets they share over the total number of propagations in the given family. We find a total of twenty families where all samples in the given family that deploys multi-process propagation also share the largest intersection of target processes. In seventeen of these families this intersection makes up the entire set of process targets from the given family, and in the remaining three families, namely Sality, TinyBanker and Madangel, some samples also propagate to more targets. In the twenty families, however, only eleven have more than one sample that does multi-process propagation, meaning that in a total of six families with multi-process propagation all samples that propagate into multiple processes share the exact same targets. Sality is particularly interesting in that all six samples that do multi-process propagation execute within the same five processes, whereas four of these samples also execute in `rundll.exe` and one also executes in `sample.tmp`. Gamarue is another interesting case, where all seven samples that do multi-process propagation inject into the same process, namely `wuauclt.exe`.

6.4.2.2 Signature diversity

Malware use a variety of Windows APIs to propagate through the system, and we now present these by way of the process propagation signatures we created in our

Family	$\lvert\mathcal{M}\rvert$	(target process name, sample count), ...,	$\mathcal{I}$	$\frac{\lvert\mathcal{I}\rvert}{\lvert\mathcal{M}\rvert}$	$\frac{\lvert\mathcal{I}'\rvert}{\lvert\mathcal{T}\rvert}$
Androm	2	(sample.exe,2), (iexplore.exe,1), (uhUhRBmr.exe,1)	1	0.50	0.50
Barys	1	(explorer.exe,1), (svchost.exe,1)	1	**1.00**	**1.00**
Bitman	1	(sample.exe,1)	1	**1.00**	**1.00**
Buzus	2	(explorer.exe,2), (sample.exe,2), (iexplore.exe,2)	2	**1.00**	**1.00**
CTBLocker	3	(sample.exe,3), (iexplore.exe,3)	3	**1.00**	**1.00**
Cerber	1	(ns710D.tmp,1), (timetasks.exe,1)	1	**1.00**	**1.00**
CoinMiner	5	(sample.tmp,2), (ns9204.tmp,1), (CNminer.exe,1), (DbQZbGtq.exe,1), (WMIC.exe,1), (ns97DF.tmp,1)	1	0.20	0.43
Crowti	2	(explorer.exe,2), (svchost.exe,1)	1	0.50	0.67
Cutwail	2	(sample.exe,2), (iexplore.exe,2)	2	**1.00**	**1.00**
Dorkbot	3	(sample.exe,3), (taskhost.exe,2), (conhost.exe,2), (explorer.exe,2), (dwm.exe,2), (cmd.exe,2)	2	0.67	0.92
Dridex	3	(explorer.exe,2), (svchost.exe,1), (edg7AF9.exe,1)	1	0.33	0.50
Eldorado	1	(708D.tmp,1)	1	**1.00**	**1.00**
Emotet	9	(sample.exe,8), (explorer.exe,8), (svchost.exe,5), (iexplore.exe,4), (taskhost.exe,3), (conhost.exe,3), (dwm.exe,3), (cmd.exe,3), (winver.exe,2)	4	0.44	0.41
Fareit	1	(sample.exe,1)	1	**1.00**	**1.00**
Fynloski	2	(notepad.exe,1), (sample.exe,1)	1	0.50	0.50
Gamarue	7	(wuauclt.exe,7)	7	**1.00**	**1.00**
Kasidet	3	(explorer.exe,3)	3	**1.00**	**1.00**
Kovter	3	(sample.exe,2), (svchost.exe,2)	2	0.67	0.50
Kryptik	3	(explorer.exe,1), (dllhost.exe,1), (73C8.tmp,1)	1	0.33	0.33
Madangel	10	(sample.exe,10), (explorer.exe,2)	10	**1.00**	0.83
Midie	4	(sample.exe,2), (sample.usr,2), (iexplore.exe,1)	2	0.50	0.40
Mira	7	(xvkwym.exe,3), (raphw.exe,2), (vlfsay.exe,2)	3	0.43	0.43
Natas	9	(sample.exe,9), (taskhost.exe,5), (conhost.exe,5), (explorer.exe,5), (cmd.exe,5), (dwm.exe,5), (ahos.exe,1), (uluwo.exe,1), (zaar.exe,1), (yros.exe,1), (loid.exe,1)	5	0.56	0.77
Ngrbot	1	(explorer.exe,1)	1	**1.00**	**1.00**
Nimnul	6	(iexplore.exe,2), (DesktopLayer.e,2), (sampleSrv.exe,2), (tXoPUA.exe,1), (sample.exe,1), (OZpdVg.exe,1), (uhUhRBmr.exe,1), (fuDwxVB.exe,1)	2	0.33	0.55
Nitol	5	(sample.exe,5), (iexplore.exe,5)	5	**1.00**	**1.00**
Nymaim	2	(sample.exe,2), (iexplore.exe,2)	2	**1.00**	**1.00**
Ramnit	7	(iexplore.exe,6), (explorer.exe,5), (samplemgr.exe,3), (DesktopLayer.e,3), (sampleSrv.exe,2), (conhost.exe,1), (OXYbHl.exe,1), (rundll32.exe,1), (OXYbHlSrv.exe,1), (taskhost.exe,1), (dwm.exe,1), (cmd.exe,1)	3	0.43	0.35
Razy	9	(750F.tmp,1), (sample.exe,1), (79D0.tmp,1), (72FD.tmp,1), (iexplore.exe,1), (7686.tmp,1), (7425.tmp,1), (7722.tmp,1), (7119.tmp,1), (73E7.tmp,1)	1	0.11	0.20
Sality	6	(taskhost.exe,6), (conhost.exe,6), (explorer.exe,6), (dwm.exe,6), (cmd.exe,6), (rundll32.exe,4), (sample.tmp,1)	6	**1.00**	0.86
Shifu	1	(taskhost.exe,1), (conhost.exe,1), (explorer.exe,1), (dwm.exe,1), (cmd.exe,1), (dllhost.exe,1)	1	**1.00**	**1.00**
Symmi	3	(sample.exe,3), (explorer.exe,1), (svchost.exe,1), (uhUhRBmr.exe,1)	1	0.33	0.50
TeslaCrypt	1	(sample.exe,1)	1	**1.00**	**1.00**
TinyBanker	9	(winver.exe,9), (explorer.exe,9), (taskhost.exe,2), (conhost.exe,2), (dwm.exe,2), (cmd.exe,2), (sample.exe,1)	9	**1.00**	0.67
Urausy	4	(sample.exe,4), (explorer.exe,3), (svchost.exe,3)	3	0.75	0.90
Ursnif	3	(explorer.exe,3), (cliciles.exe,1), (chtborui.exe,1), (7F9A.tmp,1), (factura.exe,1)	1	0.33	0.43
VBKrypt	2	(sample.exe,2), (taskhost.exe,2), (conhost.exe,2), (winver.exe,2), (explorer.exe,2), (dwm.exe,2), (cmd.exe,2)	2	**1.00**	**1.00**
Vawtrak	4	(explorer.exe,4), (taskhost.exe,3), (conhost.exe,3), (dwm.exe,3), (cmd.exe,3), (mainOUT-crypt,1), (svchost.exe,1)	3	0.75	0.83
Waski	1	(sample.tmp,1)	1	**1.00**	**1.00**
Zbot	3	(sample.exe,3), (explorer.exe,1), (kaylli.exe,1)	1	0.33	0.60

Table 6.8: Target processes of families with multi-process propagation. $|\mathcal{M}|$ denotes the number of samples in a given family that deploys multi-process execution. $\mathcal{I}$ denotes the number of samples that have the largest intersection of targets amongst the samples in a given family, $|\mathcal{I}'|$ denotes the number of targets shared by the samples in $\mathcal{I}$ multiplied by the size of $\mathcal{I}$ and $|\mathcal{T}|$ denotes the total amount of multi-process propagations in a given family.

analysis of all the multi-process propagations. The signatures we have created are API-specific, e.g. we consider `CreateFileW` and `CreateFileA` to be different API functions. The benefit of this is to have detailed statistics, from which we can indeed build abstractions, and the drawback is that some signatures can have minor differences to other signatures, e.g. whether the malware uses Unicode or ASCII.

We found a total of 417 injections, which is more than the 394 processes that are results of multi-process execution. The reason for this is that multiple processes may inject into the same target process, as shown in the previously discussed Natas sample in Figure 6.4 where multiple processes inject into the same `cmd.exe` process. The complete set of signatures we observed is shown in Table 6.9, which is sorted by the number of times they occur. In total, we discovered 33 different signatures and the most common way for malware to inject into other processes is using the traditional code injection technique with API calls to `OpenProcess`, `VirtualAllocEx`, `WriteProcessMemory`, `CreateRemoteThread`. This approach is used in 174 of 417 cases, corresponding to 41.7%. In contrast, we found six signatures that are only used once, and they rely mainly on a subset of the APIs from other signatures.

In general, when malware propagate through the system, it can do so by either launching a new process or migrating to an existing one. To estimate the proportion of these two we count the number of signatures that use `OpenProcess`, `CreateProcess` and `ShellExecute` to access the victim process, which is 211, 213 and 15, respectively. As such, there is a an almost even distribution between cases that open an existing process (`OpenProcess`) and launching a new process (`CreateProcess` and `ShellExecute`).

In Table 6.10, we show the per-family signature usage, the number of injections that rely on a given signature and the number of samples that use the given signature. We find 17 families that rely on a single signature and 23 families that deploy more than one signature. Sality is an interesting family in that it has a total of 41 process-propagations amongst six samples that all rely on the same signature to propagate. We observe similar trends in Dorkbot, Emotet, Natas, Ramnit, TinyBanker, VBKrypt and Vawtrak where samples extensively reuse the same signatures. We observe an opposite trend in families like Buzus, Dridex, Kovter, Kryptik, Fynloski and Ursnif that each has multiple samples performing multi-process propagation without any overlap in propagation technique. Buzus is particularly interesting in this context, where we observe two samples that combined use seven different injection techniques, and each of the signatures is only used once. In the following section, we will go into details with interesting injection techniques, and then in Section 6.4.4.2,

ID	APIs	#Total	#Samples	#Families
1	OpenProcess, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread	174	39	15
2	CreateProcessW, VirtualAllocEx, WriteProcessMemory, SetThreadContext, ResumeThread	48	28	14
3	CreateProcessW, VirtualAllocEx, ZwWriteVirtualMemory, ZwResumeThread	22	17	7
4	CreateProcessInternalW, ZwCreateSection, ZwMapViewOfSection, ZwMapViewOfSection, ZwCreateThreadEx, ZwQueryInformationProcess, ZwResumeThread	22	12	7
5	CreateProcessA, VirtualProtectEx, WriteProcessMemory, ResumeThread	16	15	5
6	WinExec	16	13	2
7	CreateFileA, WriteFile, WinExec	12	11	4
8	CreateFileA, WriteFile, CreateProcessA	11	9	4
9	ZwCreateSection, ZwMapViewOfSection, CreateProcessW, ZwMapViewOfSection, ResumeThread	10	8	2
10	CreateProcessA, VirtualAllocEx, WriteProcessMemory, CreateRemoteThread	9	9	5
11	CopyFileA, CreateProcessA	8	5	4
12	fopen, _write, fflush, CreateProcessA	7	7	1
13	CreateProcessA, VirtualAllocEx, WriteProcessMemory	7	6	2
14	CreateProcessW, ZwCreateSection, ZwMapViewOfSection, ZwMapViewOfSection, ZwResumeThread	6	4	2
15	CreateProcessA, VirtualAllocEx, WriteProcessMemory, GetThreadContext, SetThreadContext, ResumeThread	6	5	3
16	CreateFileW, WriteFile, CreateProcessW	6	4	5
17	CreateProcessA, VirtualAllocEx, WriteProcessMemory, SetThreadContext, ResumeThread	5	3	5
18	OpenProcess, ZwMapViewOfSection, ZwProtectVirtualMemory, ZwWriteVirtualMemory, CreateRemoteThread	4	4	1
19	OpenProcess, VirtualAllocEx, VirtualProtectEx, WriteProcessMemory, PostMessageW	3	3	1
20	GetShellWindow, ZwOpenProcess, ZwAllocateVirtualMemory, ZwWriteVirtualMemory, ZwProtectVirtualMemory, CreateRemoteThread	3	3	1
21	CreateProcessW, ZwCreateSection, ZwMapViewOfSection, ZwCreateSection, ZwMapViewOfSection	3	3	1
22	CopyFileW, ShellExecuteW	3	1	3
23	CreateFileW, WriteFile, CreateProcessW	3	3	1
24	CallWindowProcA, CreateProcessA, CallWindowProcA, VirtualAllocEx, CallWindowProcA, WriteProcessMemory, CallWindowProcA, GetThreadContext, CallWindowProcA, SetThreadContext, CallWindowProcA, ResumeThread	2	2	1
25	CreateFileW, WriteFile, CreateProcessA	2	2	2
26	CreateFileA, WriteFile, ShellExecuteA	2	2	2
27	CreateFileW, WriteFile, ShellExecuteExW	2	2	2
28	OpenProcess, DuplicateHandle, ZwAllocateVirtualMemory, WriteProcessMemory, CreateRemoteThread	1	1	1
29	CallWindowProcA, CreateProcessW, CallWindowProcA, VirtualAllocEx, CallWindowProcA, WriteProcessMemory, CallWindowProcA, SetThreadContext, CallWindowProcA, ResumeThread	1	1	1
30	OpenProcess, VirtualProtectEx, WriteProcessMemory, ResumeThread	1	1	1
31	OpenProcess, CreateRemoteThread, VirtualProtectEx, ZwWriteVirtualMemory, ResumeThread	1	1	1
32	CreateFileW, WriteFile, CreateFileW, WriteFile, ShellExecuteW	1	1	1
33	CreateFileA, CreateFileMappingA, MapViewOfFile, UnmapViewOfFile, CreateProcessA	1	1	1

Table 6.9: The multi-process signatures observed amongst all of the samples. For each signature, the table lists the APIs involved in the signature, the total number of times the signature was observed and the number of samples and families that uses it.

we will present further quantitative analysis of inter-family consistency in terms of propagation signatures.

6.4.2.3 Interesting case studies

Injection via continuously rewriting window procedure. We observed two injections, IDs #**24** and #**29**, that use dynamically generated code to hide the API calls involved in their respective injection. Specifically, three malware samples from the Urausy and Buzus families repeatedly overwrite a particular memory region with code that performs a single API call in order to create a chain-like structure that results in a complete injection procedure. The memory region is disguised as a Window procedure and called via the `CallWindowProcA` function. The effect of this is that the code responsible for the injection is constantly overwritten, remains in memory for a short time, and is never entirely represented in memory but only exposed in multiple temporal pieces. Minerva explicitly observes this by detecting a new wave of dynamically generated code whenever the malware calls `CallWindowProcA`. This makes it easy to detect that the memory is continuously updated because the execution waves have instructions execute in the same memory region.

Stealthy injection via ZwCreateUserProcess hooking. Another interesting code injection is ID #**13** which leverages three common API calls `CreateProcessA`, `VirtualAllocEx` and `WriteProcessMemory`, but does not have any call to functions like `CreateRemoteThread` or `ResumeThread`. In all cases where this signature occur, the target program is `C:\Windows\system32\svchost.exe` and has `dwCreationFlags` set to **`NULL`**, meaning the program is **not** started in suspended mode. This is unusual because without a call to `ResumeThread` or `CreateRemoteThread` it is, at first sight, unclear how the malware achieves code execution in the victim process. Instead, the malware achieves code execution in the victim process by patching the code of `CreateProcessA` at run time and then hijacking the Windows code that is in charge of creating a new process. It does this by hooking `ZwCreateUserProcess` to execute malicious code that will allocate and write memory to the target process, all nested inside the `CreateProcessA` call. To illustrate this, Figure 6.8 shows the call graph of samples that rely on this injection technique.

The hook is easy to detect given the output of Minerva from looking at the malware execution trace, shown in Figure 6.9. From the export extractor in Minerva we know `ZwCreateUserProcess` is located at `0x77c46a98` and given the first instruction

Family	$\lvert\mathcal{M}\rvert$	Σ	(signature ID, injection count, sample count), ...
Androm	2	3	(2, 2, 1) (17, 1, 1) (22, 1, 1)
Barys	1	1	(4, 2, 1)
Bitman	1	1	(2, 1, 1)
Buzus	2	7	(1, 1, 1) (3, 1, 1) (5, 1, 1) (10, 1, 1) (17, 1, 1) (28, 1, 1) (29, 1, 1)
CTBLocker	3	1	(2, 6, 3)
Cerber	1	2	(8, 1, 1) (11, 1, 1)
CoinMiner	5	5	(8, 2, 2) (11, 2, 1) (25, 1, 1) (26, 1, 1) (27, 1, 1)
Crowti	2	1	(4, 3, 2)
Cutwail	2	1	(2, 4, 2)
Dorkbot	3	3	(1, 12, 2) (2, 2, 2) (15, 1, 1)
Dridex	3	3	(1, 1, 1) (4, 2, 1) (33, 1, 1)
Eldorado	1	1	(7, 1, 1)
Emotet	9	6	(1, 18, 3) (2, 10, 6) (3, 2, 2) (4, 10, 5) (5, 2, 2) (10, 1, 1)
Fareit	1	1	(3, 1, 1)
Fynloski	2	2	(2, 1, 1) (10, 1, 1)
Gamarue	7	1	(9, 7, 7)
Kasidet	3	1	(19, 3, 3)
Kovter	3	3	(3, 1, 1) (9, 3, 1) (14, 3, 1)
Kryptik	3	3	(4, 1, 1) (10, 1, 1) (16, 1, 1)
Madangel	10	2	(1, 2, 2) (6, 10, 10)
Midie	4	2	(2, 2, 1) (6, 6, 3)
Mira	7	1	(12, 7, 7)
Natas	9	4	(1, 34, 5) (3, 14, 9) (10, 5, 5) (23, 3, 3)
Ngrbot	1	1	(1, 1, 1)
Nimnul	6	6	(7, 3, 3) (8, 2, 2) (11, 2, 2) (13, 2, 2) (17, 1, 1) (22, 1, 1)
Nitol	5	1	(2, 10, 5)
Nymaim	2	1	(2, 4, 2)
Ramnit	7	7	(1, 7, 1) (5, 2, 1) (7, 1, 1) (8, 6, 6) (11, 3, 3) (13, 5, 5) (18, 4, 4)
Razy	9	3	(2, 2, 1) (7, 7, 7) (16, 1, 1)
Sality	6	1	(1, 41, 6)
Shifu	1	1	(1, 8, 1)
Symmi	3	4	(2, 2, 2) (4, 2, 1) (17, 1, 1) (22, 1, 1)
TeslaCrypt	1	1	(2, 1, 1)
TinyBanker	9	3	(1, 19, 9) (3, 1, 1) (5, 9, 9)
Urausy	4	5	(14, 3, 3) (15, 2, 2) (20, 3, 3) (21, 3, 3) (24, 2, 2)
Ursnif	3	6	(1, 1, 1) (16, 2, 1) (27, 1, 1) (30, 1, 1) (31, 1, 1) (32, 1, 1)
VBKrypt	2	3	(1, 12, 2) (3, 2, 2) (5, 2, 2)
Vawtrak	4	3	(1, 16, 3) (4, 2, 1) (26, 1, 1)
Waski	1	1	(16, 1, 1)
Zbot	3	6	(1, 1, 1) (2, 1, 1) (15, 3, 2) (16, 1, 1) (17, 1, 1) (25, 1, 1)

Table 6.10: The per-family list of signatures. Σ denotes the number of signatures used by a given family and $\lvert\mathcal{M}\rvert$ denotes the number of samples in a given family that deploys multi-process execution. The rightmost column shows the specific signatures used in each family, the number of injections that uses a given signature and the number of samples in the given family that uses this signature.

```
CreateProcessA(victim_process)
    CreateProcessInternal
    ZwCreateUserProcess
            Hijack first instruction of ZwCreateUserProcess
                LOOP:
                    VirtualAlloc
                    VirtualProtect
                    WriteProcessMemory
        ...
```

Figure 6.8: Call graph of code injection that hooks `ZwCreateUserProcess`. API calls made by the malware are shown in bold.

is turned into a trampoline to malware code, we can easily deduce that the function is hooked. The benefit of this code injection technique is potential evasion against sandboxes and manual debuggers that only monitor `CreateProcessA` calls and not nested function calls. As such, if the hooking is not observed, then specific parts of the injection will likely go unnoticed, and the threat may evade analysis.

Stealthy injection without explicit PID access. In most cases, the signatures rely on `CreateProcess(A/W)` or `OpenProcess` to initiate multi-process propagation. These functions make it is easy for an analyst or sandbox to determine the target process as the output of the functions contains the `PID` of the process they create/open. Naturally, some injection techniques avoid these functions to hide the multi-process target, and an example of this is signature #**20**. Injection #**20** retrieves a handle to the target process by calling `GetShellWindows`, opens the process using `ZwOpenProcess` and then continues with a familiar pattern of functions to initiate execution. In this way, the `PID` of the target process is never exposed in any of the functions used in the injection, which adds a level of complexity to identifying the specific propagation procedure. Naturally, Minerva follows the malware execution regardless of knowing the target process `PID`.

6.4.2.4 Code injections vs droppers

An interesting distinction is whether a multi-process propagation is due to code injection or a dropped executable. Although these two approaches are not mutually exclusive (because a sample can drop a file, launch it and then inject code into it) it is interesting to estimate how prevalent each of them is. To give such an estimate we check for each multi-process propagation if the target process corresponds

Address	instruction
0x405996	jmp `CreateProcessA(...)`
0x77c46a98	jmp 0x403447 ***Hook in ZwCreateUserProcess***
0x403447	push ebp ***Hooked code***
...	... ***Hooked code***
0x4013c0	lea edi, [ebp-0x54] ***Return from CreateProcessA***

Figure 6.9: The malware execution trace of a code injection that hooks the function `ZwCreateUserProcess`.

Family	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{M}_D\rvert$	Family	$\lvert\mathcal{M}\rvert$	$\lvert\mathcal{M}_D\rvert$
Cerber	1	1	Ramnit	7	6
CoinMiner	5	5	Razy	9	8
Dridex	3	1	Sality	6	1
Eldorado	1	1	Urausy	4	2
Kryptik	3	1	Ursnif	3	2
Natas	9	5	Waski	1	1
Nimnul	6	5	Zbot	3	1

Table 6.11: The families with samples that propagate via dropped files. The table also shows the number of samples with multi-process propagation, denoted $|\mathcal{M}|$, and the unique number of samples in each family that deploys propagation via dropped files, denoted $|\mathcal{M}_D|$.

to a file created by the malware earlier in the execution, and if so consider it to be an execution of a dropped executable. We found 52 multi-process executions where the propagation was in a created file and 342 cases where it was not the case. As such, we estimate that 13% of the multi-process executions in our data set are due to droppers, and the remaining 87% are due to code injections. We found a total of 14 malware families that had samples propagate via created files and Table 6.11 lists these together with the number of samples in each family that propagate in this way.

6.4.3 Malware activities

We now move on to the third research question "*Are there clear relations between malicious behaviour and system-wide malware propagation?*". This research question aims to answer how malware distributes its activities across its propagation and in this section we, therefore, focus our statistics on the 151 samples that contain multi-process execution and leave the other samples out. We limit ourselves to these because our interest is in highlighting the difference between the initial process and the non-initial processes. The initial process refers to execution at process-depth one, and the non-initial processes refer to execution in processes with a depth greater than one.

6.4.3.1 Execution wave distribution

The first aspect we analyse is where in the propagation malware deploys dynamically generated code. Specifically, we are interested in knowing if malware continues to use dynamically generated code even after multi-process execution. To do this, we count the number of waves in the initial process and compare it to the number of waves in non-initial processes. Amongst the 151 samples with multi-process execution, we have a total of 545 processes with malware execution, meaning 394 of these are non-initial processes. The distribution of waves in the initial process is shown in Table 6.10, and the distribution of waves in non-initial processes is shown in Table 6.11. We notice a clear difference in their distributions. In non-initial processes, 220 out of 394 processes only have one execution wave and do, therefore, not produce any dynamically generated code. In contrast, only 41 samples of 151 in the initial processes have one execution wave, which corresponds to a drop from 56% in non-initial processes to 27% in initial processes. The average number of waves in the initial process is 3.39, where the average number of waves in non-initial processes is 1.64.

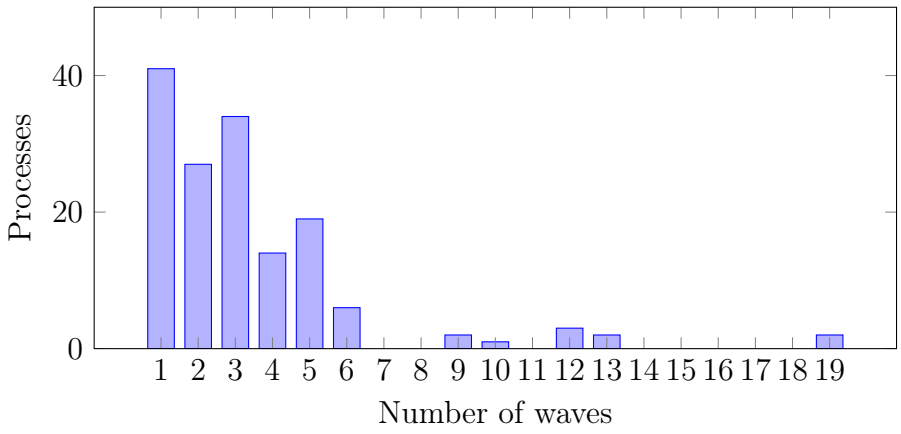


Figure 6.10: The number of execution waves in the initial process of all multi-process malware.

6.4.3.2 Code-size distribution

The second aspect we consider is how much code the malware executes in each process, and then map this into the entire context of the SPG. To measure this, we collect the number of unique instructions executed in each process and then compare the initial and non-initial processes. The reason we base our measurement on unique

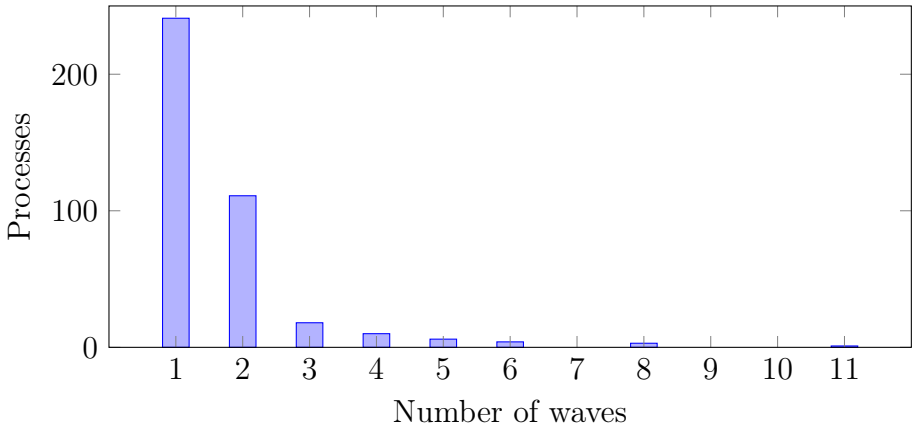


Figure 6.11: The number of execution waves in non-initial processes of all multi-process malware.

instructions and not the total number of executed instructions is to avoid bias because of decryption loops that often have a significantly larger amount of executed instructions in comparison to non-decryption loops.

Figure 6.12 shows the number of instructions executed in the initial and non-initial processes. The initial processes are significantly more dominant in terms of code size than non-initial processes with an average number of 6157 unique instructions in initial processes over 2500 in non-initial processes.

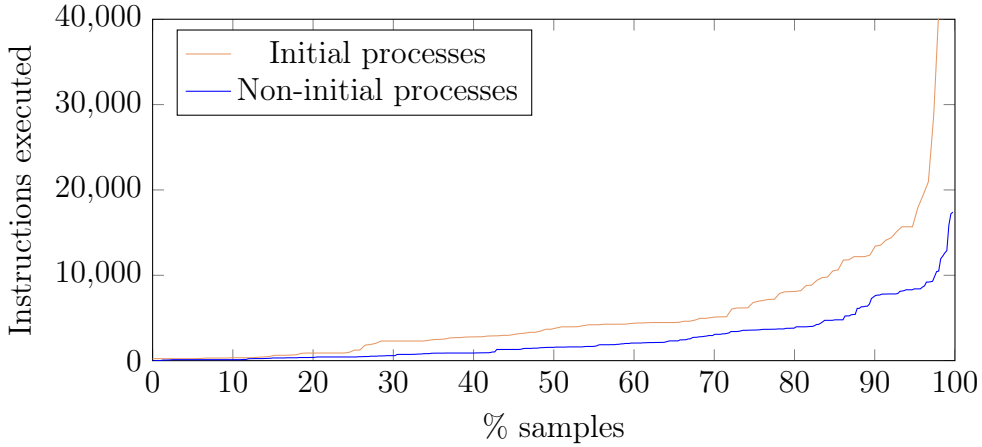


Figure 6.12: The number of instructions executed in initial processes and non-initial processes of all multi-process malware.

6.4.3.3 Sensitive API usage

Finally, we look at the distribution of suspicious API calls in samples with multi-process execution. To do this, we generate statistics about where in the SPG mal-

ware uses API functions from four groups of sensitive Windows APIs focusing on (1) registry access; (2) internet access; (3) file-system access; and (4) security via process-privilege access. We count the unique number of functions used by the malware from these groups for each process in the SPG and Figure 6.13 shows the results for initial and non-initial processes.

In general, non-initial processes have significantly more usage of APIs related to Internet and security, are on-par with registry-related APIs, but uses less APIs related to the file system. We found 89 non-initial processes call `ConvertStringSecurityDescriptorToSecurityDescriptor`, which is the most common call from this group, and only six initial processes call this function. We also observe 60 non-initial processes that use `InitializeSecurityDescriptor`, 60 processes that use `SetSecurityDescriptorDacl` and 58 processes that use `GetSecurityDescriptorSacl`. All of these functions provide features to change security descriptors of a given process, which is used for process-level privilege elevation. As such, malware is far more likely to initiate privilege escalation techniques in non-initial processes than in the initial process.

In parallel to security sensitive APIs, we see eight times more usage of Internet-related APIs in non-initial processes than in initial processes. In non-initial processes, we saw a total of 59 processes using the `WSAStartup` function, which initialises the Windows socket library (`WS2_32.dll`) and is the highest level of abstraction for working with Windows sockets. The next most popular Internet-related functions are `InternetOpen` (27 processes), `HttpOpenRequest` (22 processes) and `InternetConnect` (22 processes). In contrast, in initial processes, we only saw 11 use `WSAStartup` and zero usage of the other functions.

In Table 6.12, we show the number of samples in each family that uses functions from the four sensitive API-groups. The trends from viewing all the samples as a whole follow, and we clearly observe that families most often use Internet and security APIs in non-initial processes. In total, 29 families use Internet-related APIs in non-initial processes whereas only 6 families use them in initial processes, and 18 families use security-related APIs in non-initial processes whereas only seven families use them in initial processes. Roughly half of all samples with multi-process propagation use Internet-related APIs in their non-initial process and a little less than a third uses security-related APIs. This is far more than the usage in initial processes, which is about 7% for both API groups. We observe slightly more usage of registry-related APIs in non-initial processes and the usage of file system APIs is practically the same between initial and non-initial processes.

		Internet		Registry		File system		Security	
Family	$\|\mathcal{M}\|$	Init	Non-init	Init	Non-init	Init	Non-init	Init	Non-init
Androm	2		1	2	2	2	2		2
Barys	1		1	1		1	1		1
Bitman	1					1	1		
Buzus	2		2	1	2	2	2		
CTBLocker	3			3	3	3	3		3
Cerber	1		1	1		1	1	1	
CoinMiner	5		2		4	5	5	1	
Crowti	2		1	1		2	2		
Cutwail	2			2		2	2		
Dorkbot	3		2		2	3	2		
Dridex	3		2	2	1	3	2		1
Eldorado	1					1	1		
Emotet	9		8	5	3	7	9		6
Fareit	1		1		1	1	1		
Fynloski	2	1	1	2	1	2	1		
Gamarue	7		7		7	7	7	1	7
Kasidet	3			3	3	3	3		
Kovter	3		1	1	2	2	3		
Kryptik	3	1	2	1	1	3	3	1	1
Madangel	10			2		1	1		
Midie	4		1	1	2	4	2		1
Mira	7	1		1	7	7	7		
Natas	9		5		5	9	9		9
Ngrbot	1		1	1		1			
Nimnul	6	1	3	2	3	6	6		1
Nitol	5		3	5	1	5	3		1
Nymaim	2		1	2	1	2	1		
Ramnit	7	1	5	5	7	7	7		1
Razy	9		1	1	1	9	9		1
Sality	6	6		6	1	6	6		
Shifu	1		1		1	1	1	1	1
Symmi	3		2	1	2	3	3		1
TeslaCrypt	1			1		1	1		
TinyBanker	9		9	1	9	8	9		
Urausy	4		4	1	4	4	4		
Ursnif	3			2	2	3	3	2	3
VBKrypt	2		2		2		2		
Vawtrak	4		1	4	3	4	4	3	4
Waski	1			1	1	1	1		
Zbot	3		2	1	3	3	3		2
Total	151	11	73	63	87	136	133	10	46

Table 6.12: The amount of samples per family that use sensitive functions exposed by the Windows API. The table lists all four categories of sensitive API groups and the amount samples that use these in initial and non-initial processes, respectively.

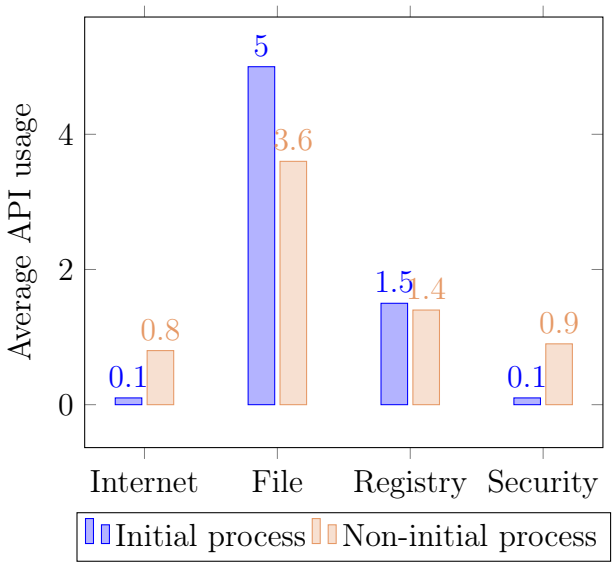


Figure 6.13: Sensitive API usage across initial and non-initial processes.

6.4.4 Propagation evolution and inter-family characteristics

We now move on to the fourth and final research question "*Has system-wide malware executions changed consistently towards one direction and are there any clear inter-family SPG consistency*". To answer this question, we first gather statistics about the yearly average of processes involved in malware execution and identify when specific propagation techniques were first used. Following this, we gather statics about inter-family consistency regarding multi-process execution and propagation techniques.

6.4.4.1 Propagation evolution

The yearly average and the standard deviation of processes involved in each malware execution are shown in Figure 6.14. The average number of processes fluctuates, and there is no clear sign towards a steady increase nor decrease. There is roughly an even distribution between the number of processes over the years, and the overall average is 1.61.

Next, we investigate how the multi-process propagation techniques have evolved by correlating the specific signatures we developed to recognise multi-process propagation in Section 6.4.2.2 to when they were first used. The number of unique signatures found each year and the number of families in our data set for each year is shown in Figure 6.15. In general, we observe a steady usage of signatures ranging from seven at the lowest (2018) to fourteen at the highest (2017). In Figure 6.16, we show the number of signatures that were discovered a given year, and, interestingly, we observe that the

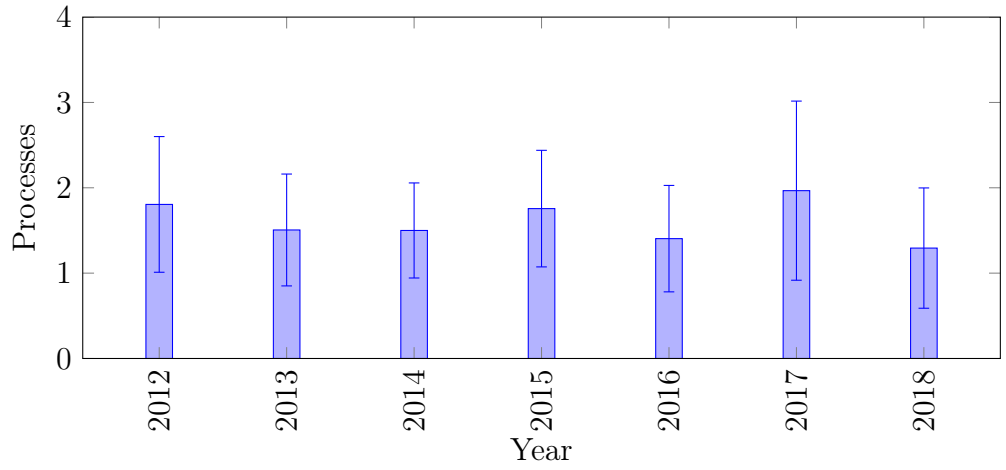


Figure 6.14: Yearly average number of processes and standard deviation of all the samples.

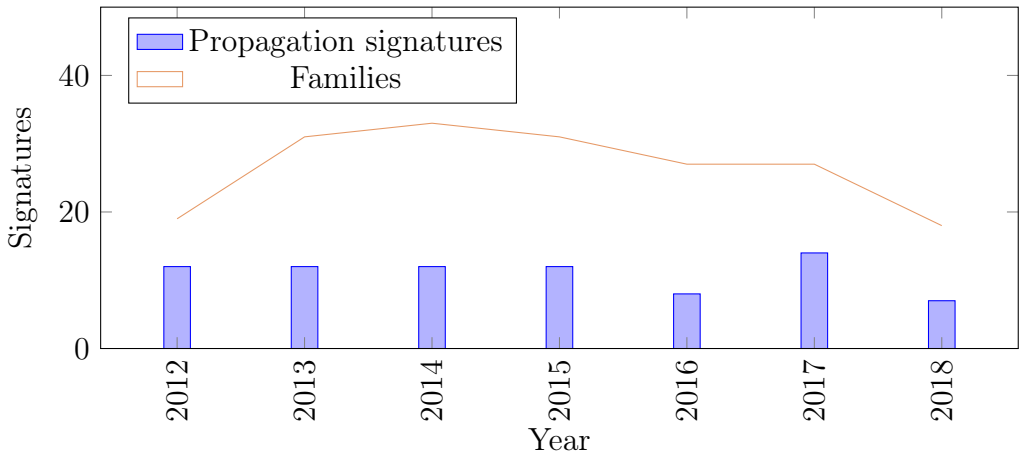


Figure 6.15: Number of different propagation signatures each year and the number of families each year.

number of novel propagation techniques is steadily decreasing. This decrease shows clearly in that 25 out of 33 injection signatures were first used in 2012, 2013 and 2014, which amounts to 75% of all injections being invented before 2015.

6.4.4.2 Inter-family propagation characteristics

Throughout this chapter, we have explored various family-related aspects of system-wide malware execution, and we now continue in this domain by giving a more general assessment of inter-family consistency using several relevant metrics. We continue with our definitions from earlier and denote the number of samples in each family as $|\mathcal{S}|$ and the number of samples that perform multi-process propagation as $|\mathcal{M}|$.

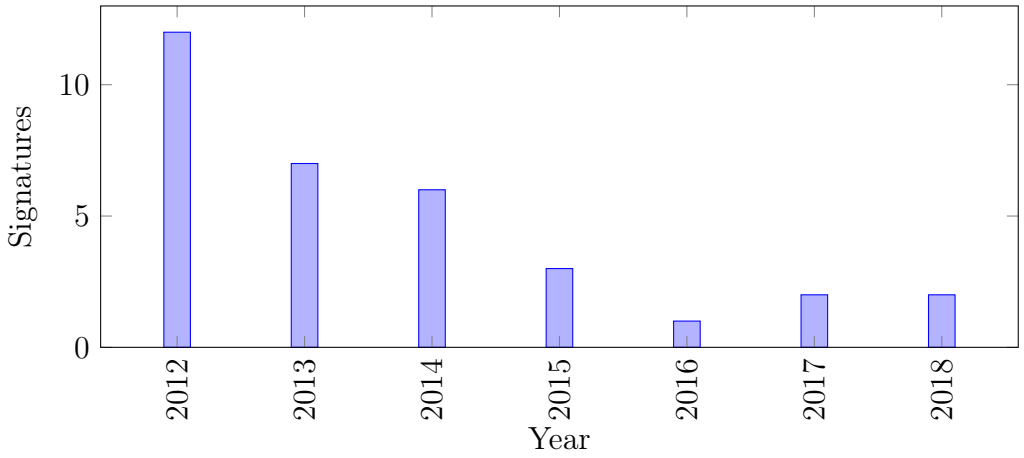


Figure 6.16: The years signatures were first used.

The first consistency measurement we determine is whether families are stable in terms of multi-process propagation. To do this, we compute the maximum number of samples in each family that either perform multi-process propagation or stays in the same process over the total amount of samples, given as $\frac{max(|\mathcal{M}|,|\mathcal{S}|-|\mathcal{M}|)}{|\mathcal{S}|}$. This is shown in the fourth column from the left in Table 6.13, and we find an overall consistency average of 0.86. We do the same check but restricted to the data set of families with samples that do multi-process propagation, i.e. $\frac{|\mathcal{M}|}{|\mathcal{S}|}$ for families with $|\mathcal{M}| \neq 0$, and the result is shown in the fifth left-most column in Table 6.13. The overall average decreases by 0.48 to 0.38. This is unexpected since it shows there is little consistency in the number of samples that perform system-wide propagation within malware families where at least one sample performs multi-process propagation.

Next, we consider the number of unique signatures deployed in each malware family, which we denote Σ. We observe an average of 2.7 different signatures in malware families that have multi-process propagating samples, as shown in the sixth column from the left in Table 6.13. Another interesting measurement is the consistency only between the malware samples that perform multi-process propagation. To quantify this, we count the number of samples in each family that has the same set of propagation signatures, and we call this Σ_{eq}. For families with multi-process propagation, the average number of samples with the same set of signatures is 2.62. To measure the inter-family consistency we then divide Σ_{eq} with the number of propagating samples from each family, given as $\frac{\Sigma_{eq}}{|\mathcal{M}|}$ and this is shown in the second column from the right. Here, we see an overall average of 0.73, meaning 73% of samples in families with propagating samples have the same set of signatures.

Finally, we look at consistency based on the maximum number of samples with

similar signatures and the number of samples without multi-process propagation, given as $\mathcal{S}_{eq} = max(\Sigma_{eq}, |\mathcal{S}| - |\mathcal{M}|)$. From $\mathcal{S}_{eq}$ we then calculate $\frac{\mathcal{S}_{eq}}{|\mathcal{S}|}$ to get a consistency measurement on the entire family based on samples with the same set of signatures or no signatures at all. On average, we see a 0.83 consistency as shown in the rightmost column of Table 6.13. This shows that 83% of samples in each family either has the same set of signatures for multi-process propagation, or no samples in the family have multi-process propagation at all. We found no families that have multi-process propagation to have a perfect score in this context.

6.5 Discussion

We now give answers to our research questions, discuss the results in more depth and also discuss the limitations of our study. We present our discussion in chronological order of the research questions.

6.5.1 Answers to research questions

Research question 1. *Is system-wide malware propagation prevalent? Yes!*
We find that 23.23% of samples perform multi-process propagation, showing that almost a quarter of malware samples rely on host-based propagation. In contrast, PaloAlto reports that 13.5% [118] of malware samples in 2013 contain code injections and Ugarte et al. [147] report that 15.6% samples (1,213 of 7,729) contain multi-process execution in a data set spanning mid-2007 to mid-2014. We believe the main reason for this is that our system is more general and captures multi-process execution that previous work miss. Another possibility for different results is data set difference, in that Ugarte et al. rely on samples that are older than ours.

System-wide malware propagation is not only prevalent in terms of multi-process propagation but also the number of dynamically generated execution waves. We find that 60% of all samples have multiple execution waves, meaning they deploy some form of dynamically generated code. In contrast, Ugarte et al. [147] found 78.7% (6088 out of 7729 samples), Codisasm [21] reports 68% and Kang et al. [83] report 98% (367 out of 374). Interestingly, we report the lowest number since our model is generic in terms of system-wide malware execution. However, our model also distinguishes malicious and benign code execution more precisely, and we suspect this to be the main reason. Previous work unreasonably over-approximates some aspects of the execution, e.g. capture dynamically generated code by any code in a process with

Family	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{M}\rvert$	$\frac{max(\lvert\mathcal{M}\rvert,\lvert\mathcal{S}\rvert-\lvert\mathcal{M}\rvert)}{\lvert\mathcal{S}\rvert}$	$\frac{\lvert\mathcal{M}\rvert}{\lvert\mathcal{S}\rvert}$	Σ	Σ_{eq}	$\mathcal{S}_{eq}$	$\frac{\Sigma_{eq}}{\lvert\mathcal{M}\rvert}$	$\frac{\mathcal{S}_{eq}}{\lvert\mathcal{S}\rvert}$
Androm	10	2	8.00	0.20	3	1	8	0.50	0.80
Artemis	10	0	10.00	—	—	—	10	—	1.00
Barys	10	1	9.00	0.10	1	1	9	1.00	0.90
Bitman	10	1	9.00	0.10	1	1	9	1.00	0.90
Buzus	10	2	8.00	0.20	7	1	8	0.50	0.80
CTBLocker	10	3	7.00	0.30	1	3	7	1.00	0.70
Cerber	10	1	9.00	0.10	2	1	9	1.00	0.90
CoinMiner	10	5	5.00	0.50	5	2	5	0.40	0.50
CosmicDuke	10	0	10.00	—	—	—	10	—	1.00
Crowti	10	2	8.00	0.20	1	2	8	1.00	0.80
Cryptlock	10	0	10.00	—	—	—	10	—	1.00
Cutwail	10	2	8.00	0.20	1	2	8	1.00	0.80
DealPly	10	0	10.00	—	—	—	10	—	1.00
Dorkbot	10	3	7.00	0.30	3	1	7	0.33	0.70
Dridex	10	3	7.00	0.30	3	1	7	0.33	0.70
Eldorado	10	1	9.00	0.10	1	1	9	1.00	0.90
Emotet	10	9	9.00	0.90	6	5	5	0.56	0.50
Fareit	10	1	9.00	0.10	1	1	9	1.00	0.90
Flood	10	0	10.00	—	—	—	10	—	1.00
Fujacks	10	0	10.00	—	—	—	10	—	1.00
Fynloski	10	2	8.00	0.20	2	1	8	0.50	0.80
Gamarue	10	7	7.00	0.70	1	7	7	1.00	0.70
Gootkit	10	0	10.00	—	—	—	10	—	1.00
Kasidet	10	3	7.00	0.30	1	3	7	1.00	0.70
Kazy	10	0	10.00	—	—	—	10	—	1.00
Kovter	10	3	7.00	0.30	3	1	7	0.33	0.70
Kraddare	10	0	10.00	—	—	—	10	—	1.00
Kryptik	10	3	7.00	0.30	3	1	7	0.33	0.70
Madangel	10	10	10.00	1.00	2	8	8	0.80	0.80
Madi	10	0	10.00	—	—	—	10	—	1.00
Mamba	10	0	10.00	—	—	—	10	—	1.00
Mazam	10	0	10.00	—	—	—	10	—	1.00
Midie	10	4	6.00	0.40	2	3	6	0.75	0.60
MiniDuke	10	0	10.00	—	—	—	10	—	1.00
Mira	10	7	7.00	0.70	1	7	7	1.00	0.70
Natas	10	9	9.00	0.90	4	4	4	0.44	0.40
Neshta	10	0	10.00	—	—	—	10	—	1.00
Neshuta	10	0	10.00	—	—	—	10	—	1.00
Ngrbot	10	1	9.00	0.10	1	1	9	1.00	0.90
Nimnul	10	6	6.00	0.60	6	3	4	0.50	0.40
Nitol	10	5	5.00	0.50	1	5	5	1.00	0.50
Nymaim	10	2	8.00	0.20	1	2	8	1.00	0.80
Otwycal	10	0	10.00	—	—	—	10	—	1.00
Padodor	10	0	10.00	—	—	—	10	—	1.00
Parite	10	0	10.00	—	—	—	10	—	1.00
Pony	10	0	10.00	—	—	—	10	—	1.00
Pronny	10	0	10.00	—	—	—	10	—	1.00
Ramnit	10	7	7.00	0.70	7	3	3	0.43	0.30
Razy	10	9	9.00	0.90	3	7	7	0.78	0.70
Renos	10	0	10.00	—	—	—	10	—	1.00
Rovnix	10	0	10.00	—	—	—	10	—	1.00
Sality	10	6	6.00	0.60	1	6	6	1.00	0.60
Shifu	10	1	9.00	0.10	1	1	9	1.00	0.90
Simda	10	0	10.00	—	—	—	10	—	1.00
Symmi	10	3	7.00	0.30	4	1	7	0.33	0.70
TeslaCrypt	10	1	9.00	0.10	1	1	9	1.00	0.90
TinyBanker	10	9	9.00	0.90	3	8	8	0.89	0.80
Urausy	10	4	6.00	0.40	5	2	6	0.50	0.60
Ursnif	10	3	7.00	0.30	6	1	7	0.33	0.70
VBKrypt	10	2	8.00	0.20	3	2	8	1.00	0.80
Vawtrak	10	4	6.00	0.40	3	2	6	0.50	0.60
Wannacry	10	0	10.00	—	—	—	10	—	1.00
Waski	10	1	9.00	0.10	1	1	9	1.00	0.90
Zbot	10	3	7.00	0.30	6	1	7	0.33	0.70
vilsel	10	0	10.00	—	—	—	10	—	1.00
Total average	10.00	2.32	0.86	0.38	2.70	2.62	8.26	0.73	0.83

Table 6.13: Inter-family malware propagation characteristics.

malware code executing and these over-approximations may capture non-malicious execution waves where we are highly resistant to this.

System-wide malware propagation has also been prevalent for many years, showing that indeed this is not a new issue. We observe a steady use of multi-process propagation throughout our entire data set, spanning 2012-2018, and we find that 40 out of a total 65 families use multi-process propagation, which is roughly two-thirds of the families in our data set. As such, system-wide propagation is prevalent not only in the number of total samples but also in the majority of families.

Research question 2. *Are the propagation strategies used in the wild diverse? Yes!*

We find that there is a range of 2 to 11 processes in which malware with system-propagation execute. As such, there is a significant difference directly in the number of processes, although 85% of samples with multi-process execution execute in either two or three processes. In addition to this, there is also a broad range of execution waves deployed by malware samples, with the majority ranging between two and 25 execution waves. We find only 13 samples with more than 25 execution waves. Previous work also reports a diversity in the number of process executions and the number of execution waves [21, 52, 83, 147].

Furthermore, we also observe diversity in process-depth, wave-depth and SPG-width. We see a maximum process-depth of five processes and a maximum SPG-width of seven processes. This means we see several malware samples that both spread out across several processes and also perform their system-wide propagation in a chain-like fashion.

One of the areas that show significant diversity in propagation strategy is the specific techniques malware use to propagate across processes. We find a total of 33 different propagation techniques, and most of these have less than ten samples using the given technique. However, many of the more rarely-seen techniques are derivatives of popular techniques. These are derivatives in various ways such as using hooking to rely on only a subset of the standard APIs. In addition to this, there is also an even distribution between propagations that inject code into existing processes and propagations that create new processes.

Interestingly, we also find that malware with samples that do system-wide propagation use on average 2.6 different propagation techniques. This also means there is diversity within the SPG of each malware sample, in the specific way they propagate through the system.

We also observe that initial processes execute more code than non-initial processes. On average, the initial processes execute about three times more unique instructions than non-initial processes. Furthermore, the set of processes that malware target to perform multi-process propagation is mainly divided between a small number of standard Windows applications.

Research question 3. *Are there clear relations between malicious behaviour and system-wide malware propagation? Yes!*
Two key observations provide arguments for the answer to our third research question. First, the number of execution waves in the initial process significantly outnumbers the number of execution waves in non-initial processes. In total, 73% of samples that deploy multi-process propagation use dynamically generated code in their initial process where the figure is 44% for processes that are a result of multi-process propagation. As such, dynamically generated code is a technique that is mostly used in the initial process but occurs in almost half of the non-initial processes.

Second, the use of sensitive API calls is a central feature that distinguishes code in the initial process and code in non-initial processes. We find far more usage of security-related APIs and Internet-related APIs in non-initial processes. This makes sense because malware uses propagation techniques, including dynamically generated code and multi-process propagation, to hide the malicious code and make the analysis of it more complicated. In addition to this, it makes sense that sensitive API calls are mainly called in non-initial processes because a large number of targets are benign Windows programs where sensitive API calls are considered less suspicious.

Research question 4. *Has system-wide malware executions changed consistently towards one direction and are there any clear inter-family SPG consistency? We find no evidence of consistent changes over the years, and for families, we find areas of both inter-family consistency and diversity!*
We find no clear evidence that the use of system-wide propagation in malware is increasing nor decreasing but rather observe a fluctuating trend. Specifically, there is no increase in either the number of processes used by malware or execution waves and there is no increase in the number of novel API patterns used to propagate to multiple processes. We find that there is a constant diversity amongst signatures used every year, but that the number of unique signatures invented is steadily decreasing.

From a high-level perspective, we find several metrics that show inter-family consistency. We find that 86% of samples in each family either propagate or do not

propagate and, closely related, we find that 83% of samples in each family follow one direction of either not deploying any system-wide propagation or implementing the same set of propagation approaches. Additionally, In 17 out of 40 families with multi-process execution, we find that the targets of the multi-process samples of each family are the same, albeit it is only six of these families that have multiple multi-process samples. In terms of propagation strategy, we find that for families with multi-process execution an average of 73% of the samples use the same APIs to propagate through the system, and we find several families where the samples in the respective family propagate using a single signature. Furthermore, we find that in 27 out of 31 families that use Internet-related APIs the samples of each family purely use functions from the API via initial or non-initial processes, meaning the samples in each family consistently use Internet-related APIs in the same way. We find a somewhat similar trend with APIs related to process-level privileges where samples from 12 families, out of a total 17, exclusively use the API from either initial or non-initial processes, albeit, 8 of these only have a single sample using the security-related APIs.

However, we also observe several metrics that show signs of inter-family diversity. For example, in the 40 families that have samples with multi-process execution, we find an inter-family average of 38% samples exhibiting multi-process execution. Furthermore, it is only in one family where all samples deploy multi-process propagation and only in 10 families that more than half of the samples deploy multi-process execution. Additionally, we find a significant variation in 9 out of all 65 families in terms of the number of execution waves deployed by the samples of the respective families, in that the standard deviation is larger than the average number of execution waves. Similarly, we find that 17 out of 40 families with multi-process execution have varying levels of process-depths, meaning the samples in each family deploy diverse propagation strategies.

6.5.2 Limitations

The main limitation of our study is that we execute the samples under one specific setting. To get broader insights about malware propagation we can leverage the use of differential studies, by analysing the malware under different contexts, similar to Cozzi et al. [44]. For example, we set the recording time to 25 seconds, and we did not perform any user stimulation. Increasing the recording time is likely to produce interesting results, and supplying user stimulation can trigger more behaviours in some types of malware. Another promising avenue is changing parameters in the execution environment, such as the state of the guest machine. For example, we used a vanilla

Windows 7 with no processes running except for the standard Windows applications. However, some malware samples rely on injecting into specific processes, such as web browsers, and the state of our guest machine does not enable this behaviour. By providing a more realistic setup, we may gather more resourceful results. In addition to this, we could also collect more comprehensive statistics by extending Minerva to support 64-bit architectures, as some malware samples are observed to deploy exotic propagation techniques only on this architecture [16].

6.6 Related work

Large-scale dynamic analysis studies. There are several existing studies based on large-scale dynamic analysis of malware. Bayer et al. performed a study [14] on samples from early 2007 to late 2008 based on their Anubis malware analysis platform. Their data set is based on submissions to their web portal, and they investigate various aspects such as file system, network and registry activity as well as botnet activity and sandbox detection. Ugarte et al. [147] present a large-scale study on packers based on an analysis platform build on top of TEMU. We have already discussed this in detail throughout the thesis and will, therefore, not elaborate here. Severi at al. presents the Malrec malware analysis system [138], which is also based on the PANDA instrumentation environment, and their study incorporates 66,301 samples spread over 1,270 families as identified by AVClass [136]. They first perform a study on how much malware modify kernel code as a means of detecting privilege escalation and then performed a study on malware classification.

Other large-scale malware analysis studies. Plohmann et al. present the Malpedia platform [121], which is a collaborative effort to gather samples and structure the malware landscape. They collect a corpus of 1800 malware samples spanning 600 families and have manually unpacked many of them. Based on the data set, they perform a study on various elements collected through static analysis such as the PE headers, control-flow analysis and API usage. Although the majority of large-scale studies focus on Windows malware, there is some work about other operating systems and architectures. A recent study by Cozzi et al. [45] performs a comprehensive investigation into Linux malware. The authors collect a data set of 10,548 Linux malware samples for more than eight different architectures and study several aspects like ELF header manipulation, persistence, deception, privilege escalation and process in-

teractions. The growing number of Android and mobile phones have also motivated several large-scale studies for malware on the Android platform [61, 146, 150, 165].

6.7 Chapter summary

In this chapter, we performed a large-scale analysis of malware in the wild, spanning 650 samples and 65 different families. Our study focused on the three aspects of system-wide malware propagation: (1) prevalence and diversity; (2) relationship to malicious behaviours; and (3) evolution and inter-family consistency.

We collected vast amounts of statistics to derive insights about malware propagation and discussed our results in detail. We found that system-wide propagation is prevalent and diverse amongst malware samples. We found clear relationships between propagation and malware behaviours and found mixed signals in terms of inter-family consistency. Surprisingly we did not see any increase in malware propagation through the years but instead observed a steady and consistent use.

The results of our study show that we can use a carefully selected data set to derive a high-level view of malware propagation, albeit this view can be blurry in places. We used this high-level view to identify key characteristics of malware propagation that has implications for existing work and also opens opportunities for new research avenues.

Chapter 7

Conclusion & future work

> "*Life is not easy for any of us. But what of that? We must have perseverance and above all confidence in ourselves. We must believe that we are gifted for something, and that this thing, at whatever cost, must be attained.*"
>
> — Marie Curie, as quoted in Madame Curie : A Biography by Eve Curie, 1937.

This dissertation explored the thesis that automated analysis of malware that propagate through the system in unconventional ways is possible, and that we can automate such analysis with a high level of precision and generality. We demonstrate this via a novel malware analysis system called Minerva.

Throughout the thesis, we maintained a thorough evaluation of our solutions based on ground-truth applications developed by ourselves and real-world malware samples from the wild. We put significant effort into evaluating our system on a broad empirical spectrum from detailed analyses to large-scale quantitative studies. We also put much effort into doing fair and meaningful comparative studies between Minerva and existing work. Our results show that there is a significant lack in support for system-wide malware analysis and that Minerva is a promising first step.

We first (Chapter 4) showed novel techniques to capture an execution trace in the context of system-wide execution and code-reuse attacks, and how to identify intrinsic characteristics of code injections within this execution trace. Our approach traces the malware execution based on information flow and does not rely on any knowledge of code injection techniques and alike. This part of our work serves as a foundation for general and precise dynamic malware analysis in the context of system-wide propagation.

Next (Chapter 5) we investigated the problem of system-wide unpacking. We built on the capabilities presented in Chapter 4 to show that automatic and accurate

unpacking of malware is possible. We introduced a novel model of dynamically generated code that is based on information flow and also techniques to precisely capture external dependencies in the packed malware. We then combined these contributions into a practical implementation in Minerva that is capable of unpacking malware so that the output is well-suited for follow-up static analysis.

In the final part of our thesis (Chapter 6), we carried out an extensive study on many diverse malware samples to assess and characterise system-wide propagation in the malware landscape. We carefully collected a comprehensive data set spanning many families, analysed each of the samples in Minerva and gathered vast amounts of statistics to derive a well-rounded perspective on system-wide malware propagation.

7.1 Lessons learned

Throughout this thesis, there are many lessons learned, and we have condensed these into several brief points, which we present below.

Implementation and measurements. Malware analysis tools are software systems. In order to derive meaningful and comparative conclusions, we must be able to test and profile these systems on extensive benchmarks.

Ground-truth data sets. Empirically analysing malware samples is laborious and time-consuming. It is often impractical and imprecise having to analyse batches of malware samples manually when testing malware analysis systems. Ground-truth data sets can significantly help with this.

Theoretical and empirical evaluations. To establish proactive techniques, it is imperative to thoroughly analyse theories conceptually and not limit analysis to empirical evaluations of existing malware samples.

Malware execution tracing. Malware execution does not follow traditional execution paradigms. It is necessary to recognise this at a fundamental level in order to build general and precise theories. Malware execution tracing must be considered on a system-wide basis, and it is fuzzy to distinguish benign and malicious code. Small changes in the analysis can have a significant impact on results, and it is, therefore, important to clearly describe how our theories and systems collect malware execution traces.

Unpacking, disassembly and dynamically generated code. Malware unpacking and malware disassembly are two closely related problems. In order to argue that unpacking is complete, it is necessary to uncover all possible malware code, and unpacking is far more than extracting dynamically generated code.

Anti-analysis techniques come in groups. Malware come with many techniques to make analysis hard. The combination of techniques enable synergies that introduce new anti-analysis techniques. It is imperative to consider anti-analysis techniques, both individually and in groups when developing defensive technologies because a combination of anti-analysis techniques can quickly break theories and tools.

7.2 Open problems and future work

The work in this thesis leaves several interesting open problems for future work. We conclude with a short list of these.

More abstractions on the malware execution trace. One of the main theoretical contributions in this thesis is the definition of malware execution trace presented in Chapter 4. This definition gives a first explicit model of malware execution tracing in a sandbox from which we define further concepts such as code injection and execution waves. However, we can start to build many more abstractions that intrinsically describe malware. As a start, a general definition of privilege-escalation seems straightforward to define based on the permissions of each instruction in the malware execution trace.

In addition to coming up with more abstractions, another exciting avenue is the comparison of such, both theoretically and practically, which would help us clearly understand trade-offs between models.

Automatically generating malware invariants. The system-wide execution that we capture can be used to extract execution-invariants by analysing a large number of samples. An interesting application of this is automatically generating host-based intrusion prevention signatures or identifying the bottlenecks in operating systems that malware leverage to execute malicious actions. The goal here is to enable complete automation from automated fine-grained malware analysis to efficient and precise endpoint protection systems. This would eliminate much manual work

and, if accurate enough, has the potential for providing better defensive systems that are easily updated.

New execution-wave models. The model of execution waves that we present in Chapter 5 is conservative because the shadow memories of two execution waves may overlap. This occurs when malware transfers execution to code from an earlier execution wave. In these cases, we include these instructions into the current execution wave rather than classifying this as the execution of a previous execution wave. The benefit of this is that we only need to maintain one execution wave per process during analysis.

An interesting area to explore is more fine-grained definitions of execution waves that constrain the overlap of shadow memories. However, it is likely that these solutions come with a more substantial performance overhead and more complexity in the implementation, which may limit the applicability of such definitions.

Another model of execution waves that is worth considering is based on continuously increasing the size of the execution waves instead of resetting them. For example, instead of assigning the shadow memory of the new execution wave to the tainted writes of the previous execution wave, we could assign the shadow memory of the next execution wave to the union of the tainted writes and the shadow memory of the previous execution wave. Specifically, replace Algorithm 5 in Section 5.3.2 with Algorithm 6. The benefit of this is only capturing new content in each execution wave.

ALGORITHM 6: dump_wave

Data: (input)Current wave $\mathcal{W}$, shadow memories $\mathcal{S}$, Tainted writes $\mathcal{T}$.
Result: Updated $\mathcal{S}$, $\mathcal{T}$, $\mathcal{W}$

1 LogInstrs(W_{pid})
2 LogTaint($\mathcal{T}$)
3 LogShadowMem($\mathcal{S}$)
4 $\mathcal{S}_{pid} \leftarrow (\mathcal{S}_{pid} \cup \mathcal{T}_{pid})$ `// Union shadow memory with tainted writes, rather than tainted writes only.`
5 $\mathcal{T}_{pid} \leftarrow \varnothing$
6 $W_{pid} = \varnothing \cup \{i\}$
7 **return** $\mathcal{S}_{pid}, \mathcal{T}_{pid}, W_{pid}$

Synthesise unpacking programs. The current approach of full system emulation to uncover the unpacking strategy of a malware sample presented in Chapter 5 is a time-consuming and resource-intensive task. An interesting next question is if the precise and detailed analysis provided by Minerva can be used to synthesise

unpacking-programs that automatically unpack malware of similar unpacking strategies without having to execute the sample. Given that such programs would be fast, they could be used in real-time to detect new variants of packed malware.

Empirically exploring the impacts of malware experimentation set up. Throughout the thesis, we have continuously performed empirical evaluations with malware samples and put significant effort into producing experiments that accurately evaluate a given phenomenon. However, during this process, we have found limited literature on how to design malware experiments and the impacts the design decisions have on the results. The work of Rossow et al. [133] is noteworthy in this context, and they provide an excellent foundation for future research in that their work mainly presents the problem from a qualitative perspective with a guide on how to design prudent experiments. As such, they do not measure the impacts and consequences that different experimental setups have on the experimental outcome, and, to the knowledge of the author, we currently have no quantitative results on this. We believe this area of research is increasingly relevant for the community to precisely measure and analyse the techniques and tools we develop, and, therefore, consider it an important avenue for future research. Minerva would be an excellent tool for this since we have extensive capabilities that allow detailed analysis of samples and also comes with tools for profiling and extracting statistics from analysis of large data sets.

Extending Minerva and our large-scale study. An interesting research direction is to extend the implementation of Minerva so the techniques can be applied in more practical settings, like Linux and Mac OSx, and potentially mobile platforms such as Android and iOS. We can then use these implementations to perform profiling of malware samples in domains other than Windows and also make an extensive comparison between malware on a variety of platforms.

Appendix A

Code injection graph of Gapz sample

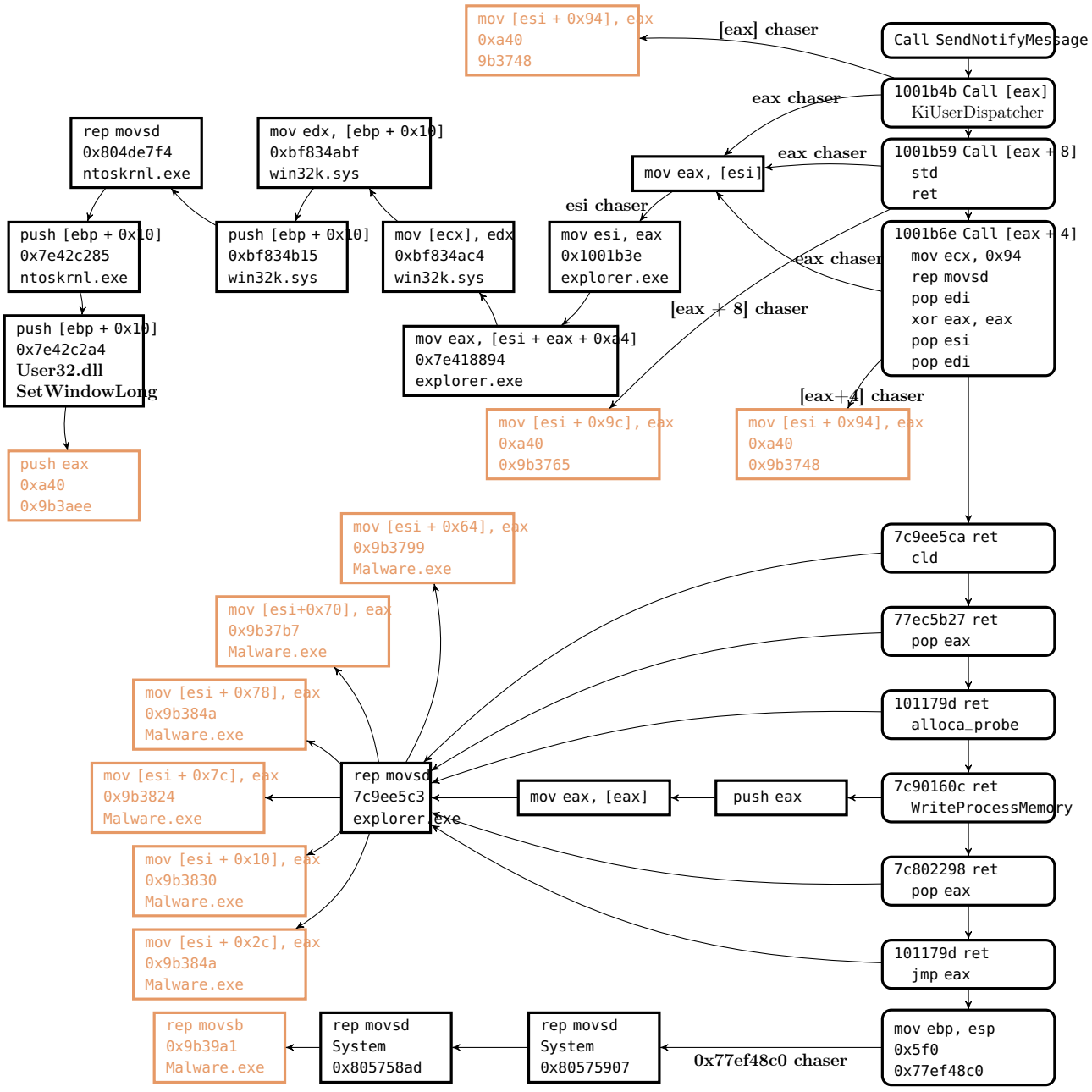


Figure A.1: Complete code injection graph of the Gapz code injection.

Appendix B

Checklists for prudent malware experimentation

Criterion	Imp.	Met	Rationale
Correct data sets			
Removed goodware	●	✔	We perform experiments with both pure malware samples and also experiments with benign samples.
Avoided overlays	●	✔	We thoroughly investigated the traces and code injections captured, and verified whether they are part of the malware samples.
Balanced families	◐	✔	We analysed four malware families with 4-9 samples from each.
Separated datasets	◐	✔	We experimented with a variety of data sets.
Mitigated artifacts/biases	◐	✦	We discussed our analysis environment thoroughly, but did not take any efforts to remove artefacts that may influence malware execution within them.
Higher privileges	◐	✔	We analyse from outside the box.
Transparency			
Interpreted FPs	●	✔	We gave a thorough discussion of false positives in Section 4.7.3.2.
Interpreted FNs	●	✔	We gave a thorough analysis showing no false negatives, see Section 4.7.3.2.
Interpreted TPs	●	✔	We gave in-depth analysis and verification of many of the true positives.
Listed malware families	◐	✔	We listed all families and samples.
Identified environment	◐	✔	We thoroughly described execution environment, see Section 4.7.2.
Mentioned OS	◐	✔	Windows 7, see Section 4.7.2.
Described naming	◐	✔	Names derived from antivirus and community consensus.
Described sampling	○	✔	We listed all samples and how we obtained them (online repositories).
Listed malware	○	✔	All samples analysed 2018.
Described NAT	○	✔	We use network simulation via INetSim, see Section 4.7.2.
Mentioned trace duration	○	✔	25 seconds.
Realism			
Removed moot samples	●	✘	Our evaluation experimented with samples that are no longer in active campaigns. However, we don't consider this to be vital here because the samples still showed the behaviour we were investigating.
Real-world FP exp.	●	✔	We performed evaluation with real malware samples and gave a thorough analysis of false positives.
Real-world TP exp.	●	✔	We gave a thorough assessment of the true positives.
Used many families	●	✔	Given our analyses are detailed we consider the number of families sufficient.
Allowed Internet	◐	✘	We provide network simulation rather than full internet access.
Added user interaction	○	✘	We do not give any user interaction. This is a limitation of our studies.
Used multiple OSes	○	✘	We conducted experiments only on Windows 7.
Safety			
Deployed containment	●	✔	We analysed in a closed network and from outside the box (malware is within virtual environments).

Table B.1: How experiments in Chapter 4 meet malware experiments guideline criteria proposed by [133]. The second column denotes the importance that [133] devotes to this subject: ● is a must, ◐ should be done, ○ nice to have. The third column describes whether we met the criteria: ✔ is met, ✦ is partially met, † is not applicable, ✘ is not met. The fourth column gives a summarised rationale for our judgement.

Criterion	Imp.	Met	Rationale
Correct data sets			
Removed goodware	●	✔	We perform experiments with both pure malware samples, verified via antivirus detection and labelling, and also experiments with benign samples.
Avoided overlays	●	✔	We performed detailed analysis of samples to confirm output.
Balanced families	◐	✔	We balanced our data sets, e.g. our study on real-world malware contains seven samples from each of the 17 families.
Separated datasets	◐	✔	We experimented with a variety of data sets.
Mitigated artifacts/biases	◐	✦	We discussed our analysis environment thoroughly, but did not take any efforts to remove artefacts that may influence malware execution.
Higher privileges	◐	✔	We analyse from outside the box.
Transparency			
Interpreted FPs	●	✔	We performed a thorough empirical study of correctness to assess the false-positives and false-negatives in Section 5.6.2 and Section 5.6.3.
Interpreted FNs	●	✔	We performed a thorough empirical study of correctness to assess the false-positives and false-negatives in Section 5.6.2 and Section 5.6.3.
Interpreted TPs	●	✔	We performed a thorough empirical study of correctness and assessed the results of our study in Section 5.6.2 and Section 5.6.3.
Listed malware families	◐	✔	We listed all families, see beginning of Section 5.6.
Identified environment	◐	✔	We thoroughly described execution environment, see Section 5.6.1.
Mentioned OS	◐	✔	Windows 7.
Described naming	◐	✔	Names derived from antivirus and community consensus.
Described sampling	○	✔	We listed all families, sometimes hashes of samples and also described how we obtained them.
Listed malware	○	✔	All samples analysed 2018.
Described NAT	○	✔	We use network simulation via INetSim, see Section 5.6.1.
Mentioned trace duration	○	✔	25 seconds recording and maximum of 120 minutes replay time.
Realism			
Removed moot samples	●	✘	Our evaluation experimented with samples that are no longer in active campaigns. However, we don't consider this to be vital here because the samples still showed the behaviour we were investigating.
Real-world FP exp.	●	✔	We performed evaluation with real malware samples and gave a thorough analysis of false positives.
Real-world TP exp.	●	✔	We gave a thorough assessment of the true positives.
Used many families	●	✔	We analysed samples from more than 17 families.
Allowed Internet	◐	✘	We provide network simulation rather than full internet access.
Added user interaction	○	✘	We do not give any user interaction. This is a limitation of our studies.
Used multiple OSes	○	✘	We conducted experiments only on Windows 7.
Safety			
Deployed containment	●	✔	We analysed in a closed network and from outside the box (malware is within virtual environments).

Table B.2: How experiments in Chapter 5 meet malware experiments guideline criteria proposed by [133]. The second column denotes the importance that [133] devotes to this subject: ● is a must, ◐ should be done, ○ nice to have. The third column describes whether we met the criteria: ✔ is met, ✦ is partially met, † is not applicable, ✘ is not met. The fourth column gives a summarised rationale for our judgement.

Criterion	Imp.	Met	Rationale
Correct data sets			
Removed goodware	●	✔	We perform experiments with both pure malware samples and verified via anti-virus detection and labelling, described in Section 6.3.
Avoided overlays	●	✔	We verified all multi-process propagations manually, described in Section 6.3.4 and Section 6.3.5.
Balanced families	◐	✔	We used the same number of samples, namely 10, from each malware family in our sample set.
Separated datasets	◐	†	Not applicable for our large-scale analysis.
Mitigated artifacts/biases	◐	✦	We discussed our analysis environment thoroughly (see Section 6.3.3), but did not take any efforts to remove artefacts that may influence malware execution.
Higher privileges	◐	✔	We analyse from outside the box.
Transparency			
Interpreted FPs	●	✔	We manually verified each of the multi-process propagation signatures.
Interpreted FNs	●	†	In this context we rely on our earlier assessment of Minerva's precision.
Interpreted TPs	●	✔	We manually verified each of the multi-process propagation signatures.
Listed malware families	◐	✔	We listed all families and the year their samples were first discovered, see Section 6.3.1.
Identified environment	◐	✔	We thoroughly described execution environment, see Section 6.3.3.
Mentioned OS	◐	✔	Windows 7, see Section 6.3.3.
Described naming	◐	✔	Names derived from antivirus and community consensus.
Described sampling	○	✔	We described families, the years they were discovered and how we obtained them (online repositories), see Section 6.3.1.
Listed malware	○	✔	We described the years samples were first identified, see Section 6.3.1.
Described NAT	○	✔	We use network simulation via INetSim, see Section 6.3.3.
Mentioned trace duration	○	✔	25 seconds recording and maximum of 120 minutes replay time, see Section 6.3.3.
Realism			
Removed moot samples	●	✘	The goal was to do a longitudinal study, and for this reason we had to rely on potentially moot samples as well. However, we due to our network simulation we ensure there will be no dead servers.
Real-world FP exp.	●	†	In this context we rely on our earlier assessments of Minerva's precision.
Real-world TP exp.	●	†	In this context we rely on our earlier assessments of Minerva's precision.
Used many families	●	✔	We include 65 families in our study.
Allowed Internet	◐	✘	We provide network simulation rather than full internet access.
Added user interaction	○	✘	We do not give any user interaction. This is a limitation of our studies.
Used multiple OSes	○	✘	We only ran on Windows 7.
Safety			
Deployed containment	●	✔	We analysed in a closed network and from outside the box (malware is within virtual environments).

Table B.3: How experiments in Chapter 6 meet malware experiments guideline criteria proposed by [133]. The second column denotes the importance that [133] devotes to this subject: ● is a must, ◐ should be done, ○ nice to have. The third column describes whether we met the criteria: ✔ is met, ✦ is partially met, † is not applicable, ✘ is not met. The fourth column gives a summarised rationale for our judgement.

Bibliography

[1] Bochs: the open source ia-32 emulation project (home page). http://bochs.sourceforge.net/, 2018.

[2] Leonard M. Adleman. Rogue programs: Viruses, worms and trojan horses. chapter An Abstract Theory of Computer Viruses, pages 307–323. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[3] Andrea Allievi and Holger Unterbrink. Cryptowall 4 the evolution continues, 2015.

[4] AMD. *AMD64 Architecture Programmer's Manual, Volume 2: Systems Programming*. AMD.

[5] Andrei Bacs, Remco Vermeulen, Asia Slowinska, and Herbert Bos. System-level support for intrusion recovery. In Ulrich Flegel, Evangelos Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 144–163, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 178–197, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. *WYSINWYX: What You See Is Not What You eXecute*, pages 202–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[8] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: learning to recognize functions in binary code. In *Proceed-*

ings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014., pages 845–860, 2014.

[9] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy: Detecting host-based code injection attacks in memory dumps. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, pages 209–229, 2017.

[10] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Bee master: Detecting host-based code injection attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings*, pages 235–254, 2014.

[11] Thomas Barabosch and Elmar Gerhards-Padilla. Host-based code injection attacks: A popular technique used by malware. In *9th International Conference on Malicious and Unwanted Software: The Americas MALWARE 2014, Fajardo, PR, USA, October 28-30, 2014*, pages 8–17, 2014.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[13] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *NDSS*, 2009.

[14] Ulrich Bayer, Imam Habibi, Davide Balzarotti, and Engin Kirda. A view on current malware behaviors. In *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, LEET '09, Boston, MA, USA, April 21, 2009*, 2009.

[15] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, Aug 2006.

[16] Magal Baz and Or Safran. Dridex's cold war: Enter atombombing, 2017.

[17] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Behavior abstraction in malware analysis. In *Runtime Verification - First International*

Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, pages 168–182, 2010.

[18] Philippe Beaucamps, Isabelle Gnaedig, and Jean-Yves Marion. Abstraction-based malware analysis using rewriting and model checking. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 806–823, 2012.

[19] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[20] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[21] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 745–756, New York, NY, USA, 2015. ACM.

[22] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] Juriaan Bremer. Cuckoo sandbox: State-of-the-art automated malware analysis. `https://www.youtube.com/watch?v=NkvjDitkDpM`, 2014.

[24] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware-assisted virtualization. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, pages 207–227, 2016.

[25] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[26] David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Dawn Song, and Heng Yin. Bitscope: Automatically dissecting malicious binaries. Technical report, 2007.

[27] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[28] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000.

[29] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *ACM Transactions on Computer Systems*, pages 143–156, 1997.

[30] Edouard Bugnion, Scott W. Devine, and Mendel Rosenblum. System and method for virtualizing computer systems, september 1998. U.S. patent 6,496,847.

[31] Alexei Bulazel and Bülent Yener. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ROOTS, pages 2:1–2:21, New York, NY, USA, 2017. ACM.

[32] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[33] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[34] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In Diego Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[35] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. In *Proceedings of the 2009 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections*, EVT/WOTE'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[36] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 167–180, New York, NY, USA, 2010. ACM.

[37] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[38] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[39] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

[40] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. 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*, ESEC-FSE '07, pages 5–14, New York, NY, USA, 2007. ACM.

[41] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, July 1995.

[42] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

[43] Microsoft Corporation. Hypervisor top-level functional specification: Windows server 2008 r2, 2008.

[44] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 870–884.

[45] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding Linux malware. In *S&P 2018, 39th IEEE Symposium on Security and Privacy, May 21-23, 2018, San Francisco, CA, USA*, San Francisco, UNITED STATES, 05 2018.

[46] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 221–232, Dec 2004.

[47] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Tainting is not pointless. *SIGOPS Oper. Syst. Rev.*, 44(2):88–92, April 2010.

[48] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 401–416, Berkeley, CA, USA, 2014. USENIX Association.

[49] Zhui Deng, Dongyan Xu, Xiangyu Zhang, and Xuxiang Jiang. Introlib: Efficient and transparent library call introspection for malware forensics. *Digital Investigation*, 9:S13 – S23, 2012. The Proceedings of the Twelfth Annual DFRWS Conference.

[50] Solar Designer. Getting around non-executable stack (and fix), 1997.

[51] Cuckoo developers. Cuckoo sandbox. `https://www.cuckoosandbox.org/`, 2017.

[52] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[53] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 4:1–4:11, New York, NY, USA, 2015. ACM.

[54] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and XiaoFeng Wang. Things you may not know about android (un)packers: A systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[55] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5, 01 2005.

[56] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.

[57] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[58] Mike Van Emmerik. Signatures for library functions in executable files. 1994.

[59] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 393–407, Berkeley, CA, USA, 2010. USENIX Association.

[60] Ryan Farley and Xinyuan Wang. Codext: Automatic extraction of obfuscated attack code from memory dump. In Sherman S. M. Chow, Jan Camenisch, Lucas C. K. Hui, and Siu Ming Yiu, editors, *Information Security*, pages 502–514, Cham, 2014. Springer International Publishing.

[61] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[62] Peter Ferrie. Attacks on virtual machine emulators. 2007.

[63] Halvar Flake. Structural comparison of executable objects. In *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop,*

DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings, pages 161–173, 2004.

[64] Volatility Foundation. Volatility - open source memory forensics. http://www.volatilityfoundation.org/.

[65] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, Apr 2013.

[66] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, 2014. USENIX Association.

[67] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. Ropmemu: A framework for the analysis of complex code-reuse attacks. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 47–58, New York, NY, USA, 2016. ACM.

[68] Pin Yahoo Groups. Failure to instrument process tree. https://groups.yahoo.com/neo/groups/pinheads/conversations/topics/12019, 2015.

[69] Fanglu Guo, Peter Ferrie, and Tzi-cker Chiueh. A study of the packer problem and its solutions. In Richard Lippmann, Engin Kirda, and Ari Trachtenberg, editors, *Recent Advances in Intrusion Detection*, pages 98–115, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[70] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[71] HASHEREZADE. https://github.com/hasherezade/demos, 2016.

[72] Andrew Henderson, Lok-Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Trans. Softw. Eng.*, 43(2):164–184, February 2017.

[73] Alex Ho, Michael A. Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. volume 40, pages 29–41, 10 2006.

[74] Ashkan Hosseini. Ten process injection techniques: A technical survey of common and trending process injection techniques. https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process, 2017.

[75] Xin Hu, Sandeep Bhatkar, Kent Griffin, and Kang G. Shin. Mutantx-s: Scalable malware clustering based on static features. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 187–198, Berkeley, CA, USA, 2013. USENIX Association.

[76] Thomas Hungenberg and Matthias Eckert. http://www.inetsim.org/, 2018.

[77] Kyriakos K. Ispoglou and Mathias Payer. malwash: Washing malware to evade dynamic analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, 2016. USENIX Association.

[78] Grégoire Jacob, Hervé Debar, and Eric Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 81–100, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] Grégoire Jacob, Eric Filiol, and Hervé Debar. Malware as interaction machines: a new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–250, Aug 2008.

[80] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 1–8, New York, NY, USA, 2011. ACM.

[81] S. Josse. Malware dynamic recompilation. In *2014 47th Hawaii International Conference on System Sciences*, pages 5080–5089, Jan 2014.

[82] Sébastien Josse. Secure and advanced unpacking using computer emulation. *Journal in Computer Virology*, 3(3):221–236, Aug 2007.

[83] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, pages 46–53, New York, NY, USA, 2007. ACM.

[84] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.

[85] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[87] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM.

[88] Thomas Kittel, Sebastian Vogl, Julian Kirsch, and Claudia Eckert. *Counteracting Data-Only Malware with Code Pointer Examination*, pages 177–197. Springer International Publishing, Cham, 2015.

[89] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[90] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 285–296, New York, NY, USA, 2011. ACM.

[91] David Korczynski. Repeconstruct: reconstructing binaries with self-modifying code and import address table destruction. In *IEEE 11th International Conference on Malicious and Unwanted Software, MALWARE 2016, Fajardo, PR, USA, October 18-21, 2016*, pages 31–38. IEEE Computer Society, 2016.

[92] David Korczynski. A characterisation of system-wide propagation in the malware landscape, 2019.

[93] David Korczynski. Precise system-wide concatic malware unpacking, 2019.

[94] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse attacks. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1691–1708. ACM, 2017.

[95] Christopher Kruegel. `https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/`, 2014.

[96] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 207–226, Berlin, Heidelberg, 2006. Springer-Verlag.

[97] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[98] Peter Kruse. W32.tinba (tinybanker) the turkish incident, 2012.

[99] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 12–12, Berkeley, CA, USA, 2010. USENIX Association.

[100] Bum Jun Kwon, Jayanta Mondal, Jiyong Jang, Leyla Bilge, and Tudor Dumitras. The dropper effect: Insights into malware distribution with downloader graph analytics. In *ACM Conference on Computer and Communications Security*, 2015.

[101] LastLine. `https://lastline.com`, 2018.

[102] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[103] Corrado Leita, Ulrich Bayer, and Engin Kirda. Exploiting diverse observation perspectives to get insights on the malware landscape. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, pages 393–402, 2010.

[104] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, fidelity and stealth in the drakvuf dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 386–395, New York, NY, USA, 2014. ACM.

[105] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects, 2019.

[106] Tal Liberman. Atombombing: Brand new code injection for windows, 2016.

[107] Tal Liberman. Bsidessf 2017, atombombing: Injecting code using windows' atoms. `https://www.youtube.com/watch?v=9HV69QGiBAU`, 2017.

[108] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 338–357, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[109] W. Liu, P. Ren, K. Liu, and H. x. Duan. Behavior-based malware analysis and detection. In *2011 First International Workshop on Complexity and Data Mining*, pages 39–42, Sept 2011.

[110] Wayne Low. Code injection via return-oriented programming. *Virus Bulletin*, 2012.

[111] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin:

Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[112] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 431–441, Dec 2007.

[113] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35, Beijing*, page 32, 2008.

[114] Monnappa22. Hollowfind. `https://github.com/monnappa22/HollowFind`.

[115] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, pages 231–245, May 2007.

[116] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. A virtual machine based information flow control system for policy enforcement.

[117] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[118] PaloAlto Networks. The modern malware review, 2013.

[119] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software., 02 2005.

[120] Giulio De Pasquale. `https://github.com/peperunas/injectopi`, 2017.

[121] Daniel Plohmann, Martin Clauß, and Elmar Padilla. Malpedia: A collaborative effort to inventorize the malware landscape. *The Journal on Cybercrime Digital Investigations*, 3(1):1–19, 2017.

[122] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. Measuring and defeating anti-instrumentation-equipped malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 73–96, Cham, 2017. Springer International Publishing.

[123] Michalis Polychronakis and Angelos D. Keromytis. Rop payload detection using speculative code execution. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software*, MALWARE '11, pages 58–65, Washington, DC, USA, 2011. IEEE Computer Society.

[124] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[125] Georgios Portokalidis and Herbert Bos. Sweetbait: Zero-hour worm detection and containment using low- and high-interaction honeypots. *Computer Networks*, 51(5):1256–1274, 2007.

[126] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[127] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[128] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. Detecting system emulators. In *International Conference on Information Security*, pages 1–18. Springer, 2007.

[129] Symantec Security Response. W32.ramnit analysis, 2015.

[130] Eugene Rodionov and Aleksandr Matrosov. Mind the gapz: The most complex bootkiv ever analyzed?, 2016.

[131] Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 798–804, 2008.

[132] Christian Rossow, Christian Dietrich, and Herbert Bos. Large-scale analysis of malware downloaders. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 42–61, Berlin, Heidelberg, 2013. Springer-Verlag.

[133] Christian Rossow, Christian J. Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 65–79, Washington, DC, USA, 2012. IEEE Computer Society.

[134] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22Nd Annual Computer Security Applications Conference*, ACSAC '06, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[135] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy*, pages 317–331, May 2010.

[136] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In Fabian Monrose, Marc Dacier, Gregory Blanc, and Joaquin Garcia-Alfaro, editors, *Research in Attacks, Intrusions, and Defenses*, pages 230–253, Cham, 2016. Springer International Publishing.

[137] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, pages 144–155, 2001.

[138] Giorgio Severi, Tim Leek, and Brendan Dolan-Gavitt. Malrec: Compact full-trace malware recording for retrospective deep analysis. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, pages 3–23, 2018.

[139] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference*

on Computer and Communications Security, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[140] Monirul Sharif, Vinod Yegneswaran, Hassen Saidi, Phillip Porras, and Wenke Lee. Eureka: A framework for enabling static malware analysis. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 481–500, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[141] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993.

[142] Asia Slowinska and Herbert Bos. Pointless tainting?: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 61–74, New York, NY, USA, 2009. ACM.

[143] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[144] Wei Song, Heng Yin, Chang Liu, and Dawn Song. Deepmem: Learning graph neural network models for fast and robust memory forensic analysis. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 606–618, 2018.

[145] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[146] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android analysis techniques. *ACM Comput. Surv.*, 49(4):76:1–76:41, 2017.

[147] Xabier Ugarte-pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers.

[148] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

[149] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function hooks without code. In *Proceedings of the 21th Annual Network Distributed System Security Symposium (NDSS)*, Febuary 2014.

[150] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In Michalis Polychronakis and Michael Meier, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276, Cham, 2017. Springer International Publishing.

[151] Jinpeng Wei, Lok K. Yan, and Muhammad Azizul Hakim. Mose: Live migration based on-the-fly software emulation. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 221–230, New York, NY, USA, 2015. ACM.

[152] David A. Wheeler. `https://www.dwheeler.com/sloccount/`, 2010.

[153] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 144–163, Berlin, Heidelberg, 2013. Springer-Verlag.

[154] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002.

[155] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. *Digit. Investig.*, 10:S59–S68, August 2013.

[156] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security Privacy*, 5(2):32–39, March 2007.

[157] Carsten Willems, Ralf Hund, and Thorsten Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. 2012.

[158] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy*, pages 674–691, May 2015.

[159] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 732–744, New York, NY, USA, 2015. ACM.

[160] Lok-Kwong Yan. Transparent and precise malware analysis using virtualization: From theory to practice, 2013.

[161] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 227–238, New York, NY, USA, 2012. ACM.

[162] Lok-Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[163] Udi Yavo and Tomer Bitton. Injection on steroids: Code-less code injections and 0-day techniques, 2015.

[164] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[165] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.