Capturing Malware Propagations with Code Injections and Code-Reuse Attacks

David Korczynski
University of Oxford
University of California, Riverside
david.korczynski@cs.ox.ac.uk

Heng Yin
University of California, Riverside
heng@cs.ucr.edu

ABSTRACT

Defending against malware involves analysing large amounts of suspicious samples. To deal with such quantities we rely heavily on automatic approaches to determine whether a sample is malicious or not. Unfortunately, complete and precise automatic analysis of malware is far from an easy task. This is because malware is often designed to contain several techniques and countermeasures specifically to hinder analysis. One of these techniques is for the malware to propagate through the operating system so as to execute in the context of benign processes. The malware does this by writing memory to a given process and then proceeds to have this memory execute. In some cases these propagations are trivial to capture because they rely on well-known techniques. However, in the cases where malware deploys novel code injection techniques, rely on code-reuse attacks and potentially deploy dynamically generated code, the problem of capturing a complete and precise view of the malware execution is non-trivial.

In this paper we present a unified approach to tracing malware propagations inside the host in the context of code injections and code-reuse attacks. We also present, to the knowledge of the authors, the first approach to identifying dynamically generated code based on information-flow analysis. We implement our techniques in a system called Tartarus and match Tartarus with both synthetic applications and real-world malware. We compare Tartarus to previous works and show that our techniques substantially improve the precision for collecting malware execution traces, and that our approach can capture intrinsic characteristics of novel code injection techniques.

KEYWORDS

Malware, Taint Analysis, Security, Code Injection

1 INTRODUCTION

Today, malware remains one of the biggest IT security threats that we have to face. Although the last decade has introduced many improvements in our defences and has significantly raised the bar for malware authors to be successful, there is still an increasing number of malware incidents reported each year. Presently, the fight against malware is challenged by two core, albeit opposite, problems. On the one hand, anti-malware companies receive thousands of samples every day and each of these files must be processed and analysed in order to determine their maliciousness. On the other hand, malicious applications are often well-designed software with dedicated anti-analysis features. This makes accurate and automatic analysis of malware a true challenge. In addition to this, many of the current tools available are constructed for specific reverse engineering purposes, which makes them less applicable to fully automated procedures and more useful for manually-assisted analysis tasks.

One problem that has particularly challenged the malware research community is analysis and detection of malware propagations inside a host system. When malware executes on a host system, it *integrates* itself to the system using stealthy approaches, often motivated by evasion and privilege escalation. An important part of our defences is to identify these propagation strategies so they can be used in host-based intrusion prevention systems (HIPS).

A key aspect of malware propagation strategies is the use of *code injections*. In the context of malware, code injection is when the malware writes code to another processes on the system so as to have this code execute. When the malware does this, it effectively executes under the context of a legitimate application like white-listed processes that goes undetected by HIPS. In cases where malware relies on well-known techniques to inject the code it is easy for malware analysis systems to identify the code injection. However, recent reports have shown that novel code injection techniques can go unidentified by fine-grained malware analysis environments [29, 36] and also bypass modern-day HIPS [27, 45]. This presents a crucial challenge because false-negatives in both environments mean that malware can operate without detection for a potentially long time.

Traditional code injection techniques rely on fixed API calls such as *WriteProcessMemory* and *CreateRemoteThread* where recent approaches have started adopting exploit-like features such as code-reuse attacks [2, 27, 29, 45]. This means, instead of writing code to another process using *WriteProcessMemory* and creating a thread in the target process with *CreateRemoteThread*, malware will, for example, write memory to a global buffer, force the target process to overwrite its own stack with the memory from the global buffer, eventually resulting in the target process executing a ROP chain controlled by the malware. In this way the malware achieves execution in the target process even without explicitly writing memory to it [27].

The security community has previously investigated automated analysis of malware propagation and code injection. These works can be roughly divided in two groups. One group that has been

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.
© 2017 ACM. ISBN 978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/0.1145/3133956.3134099

focused on automated approaches to unpacking of malware, and another group focused on detection and analysis of code-reuse attacks.

Automated approaches to unpacking malware is a well-studied area [5, 22, 30, 40]. The main focus point of these approaches is how to uncover and extract *dynamically generated code*. Code injection is a natural extension hereof, as we can consider code injections to be dynamically generated code *across processes*. The most recent work on unpacking has indeed investigated this particular problem [5, 40]. However, the suggested approaches do not address several challenges posed by recent code injections. For example, Codisasm proposed by Bonfante et al. [5] relies on hooking *CreateRemoteThread* and *CreateRemoteThreadEx* in order to monitor code execution in a different process. This excludes their approach from any code injections that do not rely on these two API calls, of which there are many [2, 27, 29, 45]. Ugarte et al. [40] make efforts into monitoring dynamically generated code via file mappings and to shared memory sections. However, these techniques only monitor for dynamically generated code *explicitly* written by the malware, and do not consider when malware uses code-reuse attacks. Because of this, their approach will not detect malicious code that has been dynamically generated *via benign code*, of which there has recently been several cases [2, 27, 29, 45].

Similarly to automatic unpacking, automatic detection and analysis of code-reuse attacks has lately received a lot of attention [10, 17, 23, 34]. However, the problem with relying on generic methods for detecting novel attacks is simply that it is hard, and analysis of special cases seems inevitable [10, 16]. In fact, a recent demonstration of a new code injection was shown to bypass both HIPS and the many exploit-mitigations deployed by Windows [28]. This particular injection technique was adapted by malware not long after the technique was published [2] . The problem with the current tools for analysing code-reuse attacks, besides being few in numbers, is that these tools provide a local and limited view on the injection in respect to the entire malware propagation. This makes them well-suited for aiding reverse engineering but are not well-adapted for complete and automated analysis of entire malware propagations [17, 23, 34].

Because of the limitations in previous work and the problems reported by the malware community, there remains to be found a general and accurate solution to automatic analysis of malware propagations with code injections and code-reuse attacks. Before we proceed, it is natural to consider what is required by such a solution. Within the entire malware propagation, the malicious code often contains several waves of encrypted code even before performing any code injection, and potentially after a given code injection as well. These encrypted waves of code contain the malicious payload which will be used for analysing the capabilities of the malware. We therefore consider a complete solution to automated analysis of malware propagations to both capture malware execution traces across the entire operating system (OS) and also raise the execution trace into abstractions in the shape of code waves and code injections.

In this paper we describe *Tartarus*, a system for automatically capturing and analysing malware propagations with code injections and code-reuse attacks. Tartarus captures 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 Tartarus to follow malware execution in the whole OS without relying on any API hooking, and also gives Tartarus the ability to identify where code-reuse attacks occur. Tartarus further deploys two novel abstractions upon the malware execution trace in order to identify dynamically generated code and code injection techniques.

To the best of our knowledge, Tartarus is the first malware analysis system that monitors malware execution based on taint analysis in combination with code-reuse attacks. Tartarus is also the first system that captures dynamically generated code based on an information-flow model and also has the abilities to automatically identify code injections and give detailed insights about them. We have systematically tested Tartarus against several datasets that demonstrate the operational practicality of Tartarus, its relevance against todays' malware landscape as well as its improvements on previous work.

Our main contributions can be listed as follows:

- We propose a technique for complete malware tracing based on taint analysis in combination with a model of code-reuse attacks.
- We propose a novel technique based on information-flow for identifying dynamically generated code.
- We propose a fine-grained technique as a unified approach for automatically identifying code injections and also providing detailed insights about them.
- We implement our techniques in a practical system called Tartarus and evaluate it thoroughly against several datasets. Our results show that Tartarus works well in operational contexts and improves over previous work in several areas.

2 MOTIVATION AND BACKGROUND

In this section we illustrate the motivation and background for our work by introducing a running example. We describe each of the three techniques: (1) dynamically generated code; (2) code-reuse attacks; and (3) code injections, and use our running example to describe the limitations of previous work.

2.1 Motivating example

Our running example is a sample collected from the Gapz malware family and Figure 1 shows the propagation strategy the sample deploys. The entry point of the malware is given by the black circle within the *Malware.exe* process. Solid arrows present control-flow, dashed arrows present data-flow and $wave_0$ are instructions explicitly present in the *Malware.exe* binary image when first loaded. When executed, the sample first deploys one wave of dynamically generated code ($wave_1$ of *Malware.exe*). The instructions of this wave then overwrite a pointer within the legitimate Windows process `explorer.exe` using `SetWindowsLong` and hijacks control of the process with a call to `SendNotifyMessage`. The execution in `explorer.exe` is transferred to a sequence of code-reuse attacks that are responsible for writing shellcode within `explorer.exe`. The code-reuse attacks transfer execution to the newly written shellcode and the shellcode itself writes a new wave of dynamically generated code inside *explorer.exe*. Finally, this wave continues to

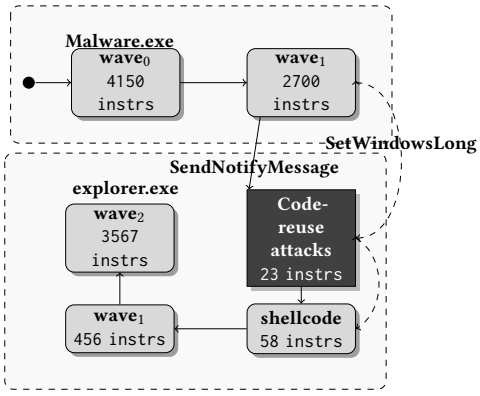


Figure 1: Malware propagation of Gapz.

deploy yet another wave of dynamically generated code with a substantially larger number of instructions.

2.2 Dynamically generated code

Nowadays, the majority of malware deploy dynamically generated code. In its simplest terms, dynamically generated code refers to when an application writes memory at runtime and then proceeds to execute this memory. Most often, malware does this by containing encrypted code inside its binary image and then decrypts this memory during execution, followed by transferring control to it.

Traditionally, dynamically generated code has been associated with the concept of packers. From a simplistic point of view, packers are applications that input a binary and output a new binary that will dynamically generate the original binary's code. The main concern of previous work in unpacking is therefore focused on automatic ways to uncover the code dynamically generated in the packed application. The techniques applied by previous solutions are fundamentally very similar. They keep a set of all the memory writes performed by the malware, and then monitor for each instruction executed in the malware process whether the bytes making up the instruction is an element of this set.

However, this heuristic is fundamentally limited because it only monitors code *explicitly* written by the malware. As such, they do not consider the cases where malware uses benign code to dynamically generate malicious code on its behalf. This limitation is visible in our motivating example where previous approaches to automatically uncovering dynamically generated code would only include $wave_0$ and $wave_1$ of the `Malware.exe` process. This is because Gapz relies on the code-reuse attacks to write the shellcode, and the shellcode is then in charge of propagating memory written by the `Malware.exe` process into the *explorer.exe* process. As a result, previous approaches will fail to recognize the code-reuse attacks and all of the malware execution that propagates from this code.

2.3 Code-reuse attacks

To hijack an application's control-flow, malware can manipulate benign code of the application even without writing any code to the application itself. We call this type of attack a code-reuse attack. One of the most popular techniques of this kind is return-oriented programming, described by Shacham [38]. The basic idea is to rely on small sequences of code that end in `ret` instructions. We call such code sequences *gadgets*. The attack then works by writing an address to memory such that whenever the target `ret` instruction is executed, it will transfer control to the address written by the adversary. By combining these gadgets in meaningful ways, the adversary can achieve complex computation and in most real-world cases even Turing complete computation [6, 8]. In the context of ROP, we call a chain of gadgets a *ROP chain*.

Code-reuse attacks is not limited to ROP, but are often leveraged with a combination of hijacking other 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 [4]. Equally, code-reuse attacks can be obtained by hijacking indirect `call` instructions, and even a mix all three instruction types. Although it is not shown in Figure 1, our motivating sample does indeed hijack both *ret* and indirect *call* instructions.

Although there exist several techniques for identifying code-reuse attacks [10, 17, 23, 34], these are designed to highlight the use of gadgets and not the overall malware propagation. As can be seen in Figure 1, the code-reuse attacks only play a very local role in the malware propagation and does not reveal much about the sample's overall structure. As such, these techniques will not be of much use in fully automatic approaches, but rather serve well in assisting manual forensics tasks.

2.4 Code injections

Code injection is when a malicious binary writes code to another process and then proceeds to execute this code. The effect of code injections is that execution of the malicious code happens inside a "legitimate" application. In practice, there are many reasons why malware use code injections. Two of the most common reasons are evasion against anti-malware solutions and escalation of process level restrictions. By nature, dynamically generated code and code injection techniques are closely related. In order to inject code, the malware writes the code at runtime, making the injected code *dynamically generated*.

Traditionally, malware has relied on a common set of approaches to inject their code. These approaches rely on well-known API calls such as *WriteProcessMemory*, *MapViewOfSection* and alike, to place malicious memory in the context of the target process, and then execute this code using API calls such as *CreateRemoteThread*, *QueueUserAPC* and *ResumeThread*. Previous work on automatic unpacking and automated malware analysis environments therefore rely on hooking these API functions [5, 11] to detect the code injections.

However, as observed in our motivating example, the malware does not rely on *WriteProcessMemory* or similar API calls to establish the code injection. In fact, the malware relies on two rather mundane API functions, *SendNotifyMessage* and *SetWindowsLong*. Furthermore, the injection technique is application-specific in that it relies on constructs specifically present in *explorer.exe*.

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 point of view. Rather, these are injection techniques that target OS-specific constructs that allows the malware to execute code in another process. The code injection is then used for achieving evasive behaviours, bypassing white-listed HIPS processes and several other purposes [28]. Effectively, the number of potential targets for these code injections is very large, because the malware is not necessarily interested in a specific set of privileged processes.

2.5 Observations and objectives

The combination of dynamically generated code, code-reuse attacks and code injections allows for complex malicious propagations that pose new challenges for malware analysis systems. Although these techniques have been treated individually in the past, they have significant overlap and malware often use the techniques in combination. Previous work suffers from focusing on one of the techniques and are therefore incomplete when the techniques are combined.

The objective with Tartarus is to construct a unified approach for automatic analysis of malware that combines all of the three techniques: (1) Dynamically generated code; (2) Code-reuse attacks and (3) Code injections. We achieve this by dividing the objective of Tartarus into two main goals:

(1) A novel approach to malware execution tracing that is tailored analysis of malware propagations. This requires a technique that is general enough to tracing malware even in the context of code injections and code-reuse attacks.
(2) Techniques for raising the collected execution trace into higher level semantics suitable for fully automatic analysis, namely dynamically generated code and code injections. Dynamically generated malicious code should be recognized independent of who wrote the code and code injections must be identified even if the injection techniques are unknown prior to analysis.

3 SYSTEM-WIDE MALWARE TRACING

In this section we present Tartarus' first goal of optimizing the completeness and precision of malware execution tracers. We start by presenting a model that allows us to reason precisely about malware execution tracers and then proceed to present our approach. At the end of the section we give a brief discussion on limitations as well as a comparison to previous work.

3.1 Abstract model of execution environment

We consider execution at the machine instruction level and our model is extended from work done by Dinaburg et al [12]. 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 then denote all possible instructions as I, where each instruction can be considered a machine recognizable 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. When a program executes, there are often many other programs executing on the system as well, 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 what 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 etc.

3.2 Malware execution trace

We now introduce the concept of *malware execution trace*, which describes what instructions of a whole-system execution is part of the malware execution. 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 gaining privilege escalation by using dynamically generated code, code-reuse attacks and code injections. As such, the execution trace of the malware may contain instructions that are not members of the program'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 function $\delta_A : S \times I \rightarrow S$ to represent the updating of a shadow memory when an instruction is executed. We call this the *propagation function*. 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 analyser 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:*

- $\Pi_A \subseteq T(P,E)$;
- $\exists v \mid m_j = i_v \wedge \Lambda_A(s_v, i_v)$ *for* $0 \leqslant j \leqslant d$.

The above definition gives us a starting point from which we can reason about the properties of malware tracers. In particular, for a given malware tracer it highlights the propagation function, δ_A, together with the inclusion predicate, Λ_A, to be the defining parts. Given two malware tracers that are targeted the same execution environment, such as X86, we can then make a detailed comparison about the instructions each analyser includes in its

malware execution trace. We do this with Tartarus and previous work [5, 40]. However, before this comparison is possible, we must first introduce our approach and we do this next.

3.3 Overview of malware execution tracing

Algorithm 1 gives an overview of our approach. For notation, let $i[A]$ denote the address of an instruction and $i[O]$ the output of an instruction. The first step **(line 2)** is to taint the memory making up the malware, and we describe this in Section 3.4. Next, we execute the malware and continue execution until there is no more taint or a user-defined timeout occurs.

For each instruction we check if the memory making up the instruction is tainted **(line 8)** and if so, include it in the malware execution trace. If it is not tainted **(line 10)**, we check if the instruction is part of a buffer that holds the code-reuse instructions we need to monitor (**line 13**). If it is, then we set a temporal variable indicating if the instruction should be included in the malware execution trace, and also remove it from the set of code-reuse to monitor. Next **(line 17)**, we check if the instruction initiates any code-reuse (**Section 3.5**). If it does, then we set the temporal variable indicating the instruction must be appended to the malware execution trace and also include the code being reused into our code-reuse buffer **(Section 3.5)**. Finally, we execute the instruction and propagate taint **(line 24)**.

Algorithm 1: Main algorithm

```
   Data: Malware execution trace Π, gadgets 𝒢.
   Result: (input) Malware sample B
1  // Initialisation;
2  𝒯 ← init_taint(B); // Taint set
3  𝒯𝒢 ← ∅;
4  // Begin full system instrumentation;
5  i ← first_instr();
6  while 𝒯 ≠ ∅ do
7      // is the instruction tainted?;
8      if i[A] ∈ 𝒯 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         // does it initiate code-reuse?;
17         if initiates_code_reuse(i, 𝒫) then
18             append = true
19             𝒯𝒢 ← get_code_reused(i, 𝒯𝒢)
20             𝒢 ← 𝒯𝒢 ∪ 𝒢
21         if append = true then
22             Π ← Π^i
23         append = false
24     (i, 𝒯) ← update(i, 𝒯);
25 return (Π, 𝒢)
```

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

Table 1: Conditions for identifying $\mathcal{GI}$ instructions.

3.4 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 memory that makes up an instruction is tainted or the instruction is part of a code-reuse attack. The only way new taint is introduced in the system apart from the initial taint is by instructions that are already tainted. The initial taint is therefore seed for the rest of the analysis and will have a large impact on the completeness and precision of the analysis.

To ensure completeness, the initial taint must cover all memory that belongs to the malware codebase, and also 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 dynamically generate code and our monitor will miss out when this code is executed. On the basis that malware can contain code anywhere in its module we taint the entire module of the malware. The tainting occurs when the program has been loaded into memory so as to ensure our tainting happens before any of the malicious code is executed.

3.5 Code-reuse identification

In Section 2.3 we introduced the different types of code-reuse attacks. The similarity between the techniques is that an indirect branch instruction in benign code redirects execution to other benign code, and the value that determines the destination is controlled by the adversary. The difference between the techniques is then 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 own purposes.

We identify $\mathcal{GI}$ instructions as dynamic 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 1 illustrates the specific rules we use to determine if an instruction is a $\mathcal{GI}$.

When a $\mathcal{GI}$ instruction has been identified, 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 $\mathcal{GC}$ can in practice include

an arbitrary number of instructions and arbitrary structure, and can vary between small gadgets and entire functions. To this date, 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. 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.

When a pair $(\mathcal{GI}, \mathcal{GC})$ has been identified we must include them in the malware execution trace. However, the gadget is only a one-time execution so we can't 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 all the instructions of the $\mathcal{GC}$ basic block is put 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 ensure only including the gadgets at the specific instance where the malware makes use of it.

We notice here that there is one exception to the rule, which is when we observe a chain of code-reuse attacks where the last hijacked indirect branch branches to tainted memory. This is seen in code injection techniques where a chain of code-reuse attacks is used by the malware to transfer execution to shellcode. In this case, the last hijacked indirect branch will effectively transfer execution to tainted memory, which means it does not satisfy the conditions for a code-reuse attack. To circumvent not including this hijacked indirect branch in the malware execution trace, if we observe a chain of code-reuse attacks followed by another hijacked indirect branch that transfers execution to tainted memory, then we also include this hijacked indirect branch in the malware execution trace as a code-reuse attack.

3.6 Propagation function

In Section 3.2 we describe that one of the key aspects of a malware execution tracer is the propagation function. The propagation function is in charge of updating the shadow memory. The core part of our propagation function is defined by our *update* algorithm, shown in Algorithm 2. We first apply taint propagation for a given instruction, done by the *propagate_taint* function. In practice, we do this with a bitwise tainting and do not rely on pointer tainting. Because our implementation is based on the DECAF [21] platform we taint directly on the QEMU tcg instructions [3]. The taint propagation rules and the soundness and precision of them are verified by Lok et al. here [43].

When the taint propagation has been done, we continue to execute (emulate) the instruction. If the instruction just executed is part of the tainted memory then we also taint the output of the instruction. We do this because some malware generates dynamically generated code without the code explicitly originating from its own memory. It is for example possible for malware to read benign code and modify this to suit its own needs and our running example of the Gapz malware employs this type of behaviour.

3.7 Comparison to previous work

The definition of malware execution trace given in Definition 1 allows us to rigorously compare Tartarus to previous works [5, 40].

Algorithm 2: update

```
Data: Instruction i, taint set T.
Result: Taint set T
// Initialisation;
T ← propagate_taint(i, T);
i_next ← exec_instr(i);
if i[A] ∈ T then
    for o ∈ i[O] do
        T ← T ∪ {o};
return (i_next, T);
```

Previous work on automatic unpacking updates the shadow memory with any memory written by instructions already inside of the shadow memory. By contrast, our approach updates the shadow memory based on our taint propagation and any output of instructions that are in the shadow memory. As such, our updates to the shadow memory includes the same as those of previous work but also propagations of shadow memory performed by instructions that are not part of the shadow memory itself. Furthermore, previous work on automatic unpacking only includes instruction in the execution trace if the memory making up the instructions are part of the shadow memory. By contrast, our approach includes instruction in the execution trace if they are either part of the shadow memory or code-reuse attacks. As such, our approach also includes the same and more instructions in the malware execution trace. This is easy to show formally and in Appendix A we give a formal treatment that shows we indeed capture the same instructions and also more, compared to previous work.

In this section, we have shown how Tartarus achieves one of its two purposes, namely a general approach to malware execution tracing in the context of dynamically generated code, code injections and code-reuse attacks. We now proceed to consider how Tartarus achieves its second goal of raising the execution into higher-level semantics.

4 CODE WAVES

The first abstraction we propose on the malware execution trace is a novel approach to identify dynamically generated code. The goal of uncovering dynamically generated code is to identify whenever memory that was written during runtime is also executed. The approach by previous work is to divide the malware execution into code waves such that each code wave consists of memory explicitly written by the previous code wave [5, 24, 40] ([40] uses the terminology *layers* instead of code waves). However, we cannot rely on this approach because it can not identify malicious code that was dynamically generated *via benign code*, i.e. the malware triggered benign code to do the writing of the dynamically generated code. Instead, Tartarus takes a different approach and raises the malware execution trace into dynamically generated code waves independent of who wrote the code, but on the basis that the written code must originate from the malware.

We model dynamically generated code on a process-level basis and consider the first code wave to be the malware module when first loaded into memory. To identify code waves we use a per-process shadow table that is a memory snapshot taken at one point

of the tainted memory in some process, and each shadow table corresponds to one code wave. When a malicious instruction executes, we can then map the bytes making up the instruction with the bytes of the shadow table. If we observe any inconsistencies with the shadow table and the currently executing instruction, we can conclude this instruction is part of a new wave of dynamically generated code.

Because the shadow table is composed of tainted memory and tainted memory is propagated through both benign and malicious instructions, then we effectively define dynamically generated code independent of who wrote the code. However, since the tainted code originates from the malware itself, then we can effectively declare the dynamically generated code to be dynamically generated *malicious* code. These are the two aspects that separate our technique from previous work and allows our technique to be more general. The ability to identify dynamically generated code in this way is vital for identifying dynamically generated code across processes because in most cases memory written from one process to another is not done by the instructions explicitly part of the malware but rather by some OS provided APIs or features.

In practice a shadow table is a hashtable where each key is an address in virtual memory and the corresponding value is the byte located at that address when the pair was inserted into the hashtable. Whenever a tainted instruction is executed we have one of four possible cases:

(1) the instruction is inside a process that does not have a shadow table assigned;
(2) the instruction is inside a process with a shadow table assigned, but the address of the instruction is not in the shadow table;
(3) the instruction is inside a process with a shadow table and the address of the instruction is inside the shadow table, but the memory in shadow table is not similar to the instruction executed;
(4) the instruction is inside a process that has a shadow table and the address of the instruction is in the shadow table, and the memory in the shadow table is similar to the current instruction.

In all cases but (4), we consider the instruction to be entrypoint of a new code wave. Further, cases 1-3 each indicate a high-level feature of the new code wave. In case (1) we have an instance of code injection, in case (2) we simply have a new code wave and in case (3) we have a new code wave that has overwritten memory in a previous code wave.

When an entrypoint of a new code wave is discovered, Tartarus first clears the shadow table of the given process, and if a shadow table does not already exist Tartarus simply initiates one. Next, the entire process memory is scanned and every byte that is tainted is put in the new shadow table. After the scanning, the shadow table contains the current code wave of the process under execution.

5 CODE INJECTIONS

The second abstraction that we propose is to identify code injections in the malware execution trace, and also extract essential insights about these by performing semantics-aware dependency analysis on the malware execution trace and the taint propagation log. The procedure consists of the following three steps: (1) from the malware execution trace, identify where control-flow goes from one process to another and arrange such findings into transition pairs; (2) find any code-reuse attacks involved in the transition; (3) perform backward dependency analysis on these two components and generate a code injection graph; In this section, we detail each of these steps.

5.1 Transition model

The transition model captures when the malware execution flows 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 according to when they were observed in a single cored 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. [40] 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 for this is that a context-switch does not reflect a control-flow transition from the process or thread being switched out to the one being switched in. Bonfante et al. [5] hook a set of function calls which are known to initiate execution in remote processes. Although this approach works well in practice when the injection techniques are known, it lacks generality because it cannot identify unknown code injections.

Our approach is instead to first identify all target instructions in a general manner independent of function hooking, and then trace backwards in order to identify the initiator instruction. In comparison to an approach that first identifies the initiator instruction and then the target instruction, our approach is able to identify the target instruction independently of the initiator instruction exactly because we rely on taint for tracing the malware execution. When we trace backwards to identify the initiator instruction, we then use function hooking to identify if the injection matches an already known injection technique, and if it does not we then deploy a general heuristic to identify the initiator instruction.

To identify target instructions we monitor for malware code execution inside processes where the malware has previously not executed. When we observe this behaviour, we know (1) this is the first instruction in a given code wave and (2) injection of code has occurred because this is the first time malware code is executed in the process. We label each of these instructions as target instructions. We note here that we only declare an instruction a target instruction if it is not a code-reuse attack.

When a target instruction has been found, we proceed to find the corresponding *initiator*. During execution we monitor all API calls done by instructions in the malware execution trace. This includes obfuscated calls such as `push X; rol [esp], Y; ret`.

We then 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`. When a target instruction is observed, we first check if there is an element $e \in F$ that initiates execution of this code. If there is, then we declare the element e as the initiator of the code injection. In is important to note here that the hooking is *only* used to identify the injection initiator and *not* to identify that a code injection has happened. Our approach to identifying whether a code injection has happened is completely independent of our use of hooking. 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 related instruction. This instruction is then identified as the *initiator*.

5.2 Injection mechanics

When the initiator and the target instruction pair has been found, we continue to identify if there is any code-reuse in-between them 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 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 *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 2 shows the key components of the code injection collected by Tartarus 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 then transfer control to 6 ROP gadgets. The target instruction is `mov ebp, esp` at address `77ef48c0`.

To identify code injections that *purely* rely on code-reuse attacks and not any target instruction, such as the malware proposed by Vogl et al. [41], we also identify a code injection if we observe a code-reuse sequence of longer than 10 gadgets.

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 give little insight into *how* the malware established these components. To give insight about this we construct the code injection graph. A code injection graph describes the control-flow of the code injection and the propagation of tainted memory involved in the code injection. The nodes of the graph are either instructions in the malware execution trace or taint-propagation instructions. The edges in the graph therefore show either control-flow or taint-flow. In practice we also annotate the nodes with several descriptive elements such as the modules and functions they are part of.

To construct the code injection graph we analyse the taint-propagation history of the tainted memory in the catalyst and the target. Specifically, we trace backwards on the instructions that have propagated the tainted memory until one of two conditions is satisfied: (1) the instruction propagating tainted memory is part of the malware execution trace or (2) the instruction propagating memory propagates memory from a code wave different than the code wave of which the target instruction belong.

PID	**Address**	**Instruction**
4d0	9b3b00	`call SendNotifyMessage`
5f0	1001b4b	call [eax] ; KiUserAPCDispatcher
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 2: Key components of Gapz code injection.

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

Figure 3: Code of KiUserAPCDispatcher.

6 IMPLEMENTATION

We have implemented the techniques described in the previous sections into a practical system called Tartarus. Our implementation consists of three main parts: (i) a dynamic analysis component that executes a given sample in our sandbox; (ii) a component that does analysis on the output of our dynamic analysis and (iii) a manager component that wraps around the two other components to allow for analysis of large sample sets.

The dynamic analysis component emulates the malware execution and is built on top of DECAF [21]. It is in charge of capturing the malware execution trace described in Section 3 and identifying code waves described in Section 4. In addition, it performs several supporting tasks such as dumping memory of code waves, construct the pair-wise execution order of instructions inside each code wave, collect obfuscated library calls (like those described in Section 5.1) and dump taint-logging information. The second component takes as input the output of our dynamic analysis component and identifies and analyses code injections. It also reconstructs control-flow

within each code wave based on disassembly of the code wave's memory dumps and the pair-wise execution order of instructions in the code wave. Based on the code waves and the code injections this component then generates a system-wide control-flow graph that shows the propagation strategy deployed by the malware. An important aspect to note here is that our implementation will discard malware execution inside a process if this process only has 10 or less instructions executed inside of it.

The dynamic analysis component does not need to log the entire execution trace. This is because the pair-wise execution order of instructions gives us information to reconstruct control-flow within each code wave, and identification and analyses of code injections only require API calls, code-reuse attacks and taint-logging information. Additionally, for each code injection we can construct the injection mechanics even without the taint-log because we don't need to backtrack on the taint propagations. We only need the taint-logging information to derive the entire code injection graph. As such, Tartarus comes with two settings, one that produces the taint-log and one that does not. The one that does not produces minimal outputs (a few MB) from the dynamic analysis where taint logs can become several (1-5) GBs for a large malware sample. In total, Tartarus consists of about 9000 lines of C code and 3500 lines of Python.

7 EVALUATION

To verify the effectiveness of Tartarus, we have evaluated it against a set of benchmark applications comprising synthetic applications and real-world malware. In Section 7.1 we experimentally validate the correctness of Taturatus by matching it with applications where we have ground-truth. Then in Section 7.2 we compare Tartarus with previous works and our results show that Tartarus is more precise by finding code injections in more applications. In Section 7.3 we perform in-depth analysis on two case studies, demonstrating that Tartarus is able to capture malware propagations, dynamically generated and code injections, and also give valuable insights about the three. In Section 7.4 we present a study on the performance of Tartarus relative to malware samples in our experiments, and finally in Section 7.5 we report observations from matching Tartarus with 934 recently collected PE files from online malware repositories.

7.1 Experimental validation of correctness

In our first experiment we empirically evaluate the correctness of Tartarus. To do this, we match Tartarus with applications where we know if the applications perform code injections or not. In total, Tartarus is evaluated against three data sets comprising 49 applications.

The first set, *A*, is composed of several code injection techniques that are publicly documented. Four of these techniques include code-reuse attacks and six of them do not. The second set, *B*, is composed of a set of malware samples from 4 malware families where anti-malware companies have documented that these malware samples inject code into other processes. We have only selected files from reports where hashsums are given for the malware samples to ensure we select correct samples. The third set, *C*, is composed of benchmark applications where we are sure they do not inject code into other processes. The benchmarks collected are all from the WCET benchmark suite [19].

			Tartarus		CO	CS
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) WPM, CRT	1		0	1	1	1
(A) WPM, STC, RT	1		0	1	0	1
(A) WPM, QAPC	1		0	1	0	0
(A) MVS, STC, RT	1		0	1	0	1
(A) MVS, CRT	1		0	1	1	1
(A) MVS, QAPC	1		0	1	0	0
(B) CryptoWall 4 [1]	4		0	4	0	4
(B) Gapz [36]	4	✓	3	4	1	1
(B) Ramnit [35]	12		0	12	4	7
(B) Tinba [25]	8		0	8	8	8
Total	38		7	38	15	24

Table 2: Evaluation with code injecting binaries.
#CRI = Code-reuse injection, #CI = Code Injection, #Codeless = Modified version of Atombombing that relies purely on code-reuse attacks. #WPM = WriteProcessMemory, #CRT = CreateRemoteThread, #STC = SetThreadContext, #RT = ResumeThread, #QAPC = QueueUserAPC/NtQueueUserAPC, #MVS = MapViewOfSection, #CO = Codisasm, #CS = CuckooSandbox.

		Tartarus	CO	CS
samples	# num	CI	CI	CI
(C) WCET [19]	11	0	0	0

Table 3: Evaluation with non code-injecting binaries.

The first five columns of Table 2 show the results of matching Tartarus with the samples from set *A* and *B*. As can be seen, Tartarus correctly identifies code injection in all of the code injecting binaries, and also identifies when code-reuse attacks are part of the code injection techniques.

Given that our technique relies on taint to capture malware execution, it is imperative that the taint does not explode. Tartarus finds no code injecting binaries when matched with our data set C, as shown in the first three columns of Table 3. However, these samples are fairly simple and do not behave in any complex system-interactive behaviours. To measure the number of false positives Tartarus produces in contexts where the samples use a lot of system activities we match for each malware sample in dataset B the code injections as identified by Tartarus with those identified in the anti-malware companies' reports.

In [25] CSIS and Trend Micro reports that Tinba injects into *winver.exe, explorer.exe, svchost.exe, firefox.exe* and *iexplore.exe*. In 6 of the 8 samples we analysed, Tartarus found injections into *winver.exe, explorer.exe, firefox.exe* and *chrome.exe*. In 1 of the 2 other samples Tartarus found injections into *winver.exe, explorer.exe,*

svchost.exe, firefox.exe and *chrome.exe* and finally in the last sample Tartarus found injections into 20 processes on the system. We picked 2 samples of the first 6 to confirm that the malware indeed injects into chrome.exe, and we also manually analysed the last sample to verify that the sample injects into all processes on the system for which it has privileges.

In [36] researchers from eset report that Gapz injects code into the Windows process *explorer.exe* and also uses code-reuse attacks. Tartarus found code injections into *explorer.exe* in 3 of the 4 samples we analysed and code-reuse attacks in all 3 of these. In two of these samples Tartarus captured *explorer.exe* to be the only process where injection occurred, where in the other one Tartarus captured code injections into *explorer.exe* but also *svchost.exe* and *winlogon.exe*. This sample contained about 3800 unique instructions inside *explorer.exe* and only 22 and 138 unique instructions within *svchost.exe* and *winlogon.exe*, respectively. The last sample exhibited no 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 [35] researchers from Symantec report that Ramnit injects code into *IEXPLORE.EXE* and *svchost.exe*. In 9 out of the 12 Ramnit samples Tartarus found injections into *IEXPLORE.EXE* and *svchost.exe*. In the remaining three samples, Tartarus captured code injection into *IEXPLORE.EXE* and *svchost.exe* but also 25, 20 and 17 instructions inside of *drwtsn32.exe*, respectively.

In [1] researchers from Cisco report that CryptoWall 4 injects into *svchost.exe* and *explorer.exe*. In all four samples of CryptoWall 4.0 in our data set we observed malware execution in these two processes. In two of the samples *svchost.exe* and *explorer.exe* were the only processes of which injection occurred. In the two other samples, Tartarus also captured 20 and 17 instructions inside *vssasmind.exe*, respectively.

7.2 Comparative evaluation

To assess the quality of our results, we put Tartarus in context with other approaches. Our second experiment, presented in this section, compares our solution with two approaches that are state-of-the art in malware analysis. We specifically compare our tool with a recent malware disassembler, Codisasm [5], and a coarse-grained malware analysis platform, namely Cuckoo Sandbox.

Codisasm relies on both static and dynamic analysis and is aimed at disassembling binaries with self-modifying code and overlapping instructions. The tool is built on top of PIN and hooks two API calls `CreateRemoteThread` and `CreateRemoteThreadEx` to follow the malware propagation. Although Codisasm is designed as a malware disassembler, it relies on capturing an instruction-level execution trace of the malware to perform the disassembly. It furthermore identifies dynamically generated code by monitoring for execution of memory that is explicitly written by the malware, as described in Section 4. As such, Tartarus and Codisasm both capture the execution trace and dynamically generated code, although with two different approaches, and this is the reason we select it as a comparative benchmark. CuckooSandbox is a malware analysis tool that is heavily used in industry and deploys a malware analyser with system and API call granularity [11]. It contains many techniques for automatically following malware in case of code injections. Both Codisasm and CuckooSandbox follow malware based on function hooking and heuristics about known injection techniques.

The results of executing our sample set A and B in Codisasm and CuckooSandbox is shown in column 6 and 7 of Table 2. As can be seen Tartarus outperformed both CuckooSandbox and Codisasm by a large margin. CuckooSandbox detected code injection in 24 samples and Codisasm in 15 samples of the 38 applications.

In comparison to Cuckoo Sandbox, the first thing we noticed 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 3 of the Gapz malware samples and 5 of the Ramnit malware samples. The sample from the Gapz family that Cuckoo correctly identified as containing code injection was the sample without code-reuse attacks as described above. The use of `ResumeThread` was accurately reported by both Tartarus and Cuckoo Sandbox.

The second thing we notice when comparing to Cuckoo is that Cuckoo failed to correctly identify code injections via remote procedure calls. We believe this is because many remote procedure calls do not constitute code injections. Therefore Cuckoo cannot create a hook and label each remote procedure call an injection because it will produce many false positives. Tartarus does not run into this problem because tainted code must be be executed for Tartarus to declare that a code injection occurs, and Tartarus can therefore identify *which* remote procedure calls result in a code injection. It may seem Tartarus should identify a code reuse attack for the remote procedure calls because an indirect branch in the target process transfers execution to a value set by the process sending the remote procedure call. However, in our cases, the destination of the indirect branch is *tainted code* so the conditions for a code-reuse attack, as described in Section 3.5, are not satisfied. Furthermore, it is important to note here that even in the case where a remote procedure call is performed, and not to tainted memory, then we will still not declare it as a code injection because only 1 code-reuse attack will be observed. As such, it does not satisfy the conditions for a code injection described in Section 5 because there is no target instruction nor a chain of code-reuse attacks.

Codisasm found code injections in 15 of the 36 samples. In particular, Codisasm found code injections in both of the synthetic samples that relied on *CreateRemoteThread*. Because we don't have direct access to their system, but rather through a web interface, it is difficult to assess the specific cause of limitations in the other samples. For several samples we would get get error messages back from the server that either an unknown malfunction occurred or the timeout occurred because the system had crashed or wasn't 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 calls to `CreateRemoteThread` and `CreateRemoteThreadEx`. However, there are many techniques that do not use either of these API functions to inject code. Notably only 2 out of 10 injection techniques in data set A makes use of `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

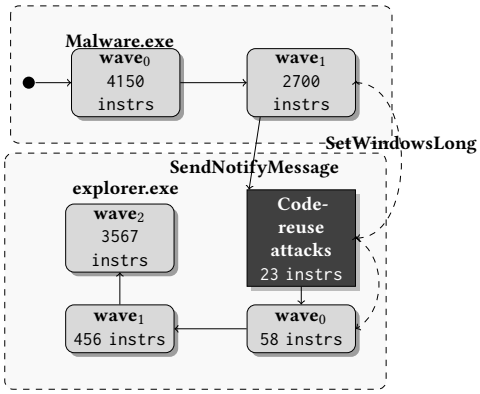


Figure 4: Malware propagation of Gapz.

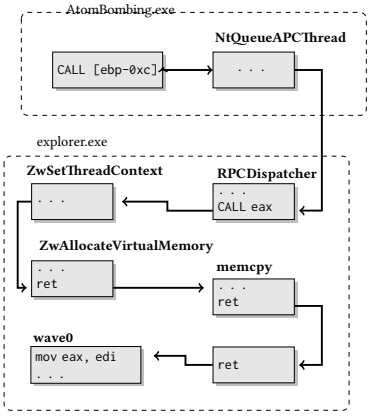


Figure 6: AtomBombing caught by Tartarus with hook on NtQueueAPCThread.

previously been reported to be unreliable when instrumenting malware. For example, it fails instrumentation in case of code injection into processes with multiple active threads [18].

7.3 Detailed analysis

To demonstrate the precision of our approach and its ability to give insights about code injection techniques we present in this section two detailed case studies of Tartarus. The first case study is from a Gapz malware sample and the second case is from a recently published code injection technique called AtomBombing.

Gapz malware. Figure 4 shows a high-level view of the system-wide CFG produced by Tartarus when matched with a sample from the Gapz malware family. The solid arrows represent control-flow, the dashed arrow represent data-flow and the black circle is the entry point.

The malware first decrypts itself with a single wave of self-modifying code and then proceeds to inject code into `explorer.exe`. Because Tartarus has identified an injection catalyst, we know there are code-reuse attacks involved in the injection. Furthermore, the injection transfers control to a rather small code wave of only 58 instructions. This suggests that the malware uses a code-reuse attack to leverage shellcode execution inside of `explorer.exe`, and this is indeed the case. The key components identified by Tartarus are shown in Figure 2, and Figure 5 shows a part of the code injection graph for the three first gadgets in the key components. Rounded boxes are control-flow and squared boxes are taint-propagating nodes. The rounded boxes shows the code-reuse initiator and also the gadget itself.

Investigating the key components, we observe the first code-reuse attack is a call to `KiUserAPCDispatcher`, and the code of this function is shown in the bottom of Figure 3. This gadget puts the value of `esp + 0x10` into `edi` and the second gadget executes the instruction `std` which will cause the direction flag to be set. 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 set to `esp + 0x10` by the first gadget, which means that the memory at the top of the stack to 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 here that the malware execution trace is *subset* of all the instructions executed. When the third gadget is executed there has in fact been several push instructions between the first and the third gadget, resulting in a larger distance between `esp` and `edi` than `0x14` as can be thought from looking at the malware execution trace. However, these are not part of the malware execution trace and is therefore not shown by Tartarus.

Investigating the code injection graph, we observe that Tartarus correctly identifies `SendNotifyMessage` as the code execution initiator and `SetWindowsLong` as a function responsible for data-flow in the overwritten addresses in the code-reuse attacks. Further analysis reveals that the ROP chain uses WriteProcessMemory to overwrite memory inside the process of `explorer.exe` and then proceeds to transfer execution to the first wave inside `explorer.exe`. Investigating the complete code injection graph, shown in Appendix A, we see with minimal effort that the code-reuse attacks does in fact turn into a ROP chain and also that the return addresses were overwritten by the `rep movsd` instruction.

AtomBombing. Recently, researchers discovered a new injection technique called *AtomBombing* [27], which uses code-reuse attacks to avoid using standard API calls for code injection. The technique was first presented 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 [2].

AtomBombing abuses the global atom table in Windows to share memory between processes and undocumented asynchronous procedure calls to force the injectee 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 injecter then uses `NtQueueApcThread` to force the injectee process to call `GlobalGetAtomName` such that the ROP chain and shellcode is stored inside the target process. To invoke execution, the injecter again uses `NtQueueApcThread` to force the injectee to call `SetThreadContext` to navigate *eip* and *esp*. *eip*

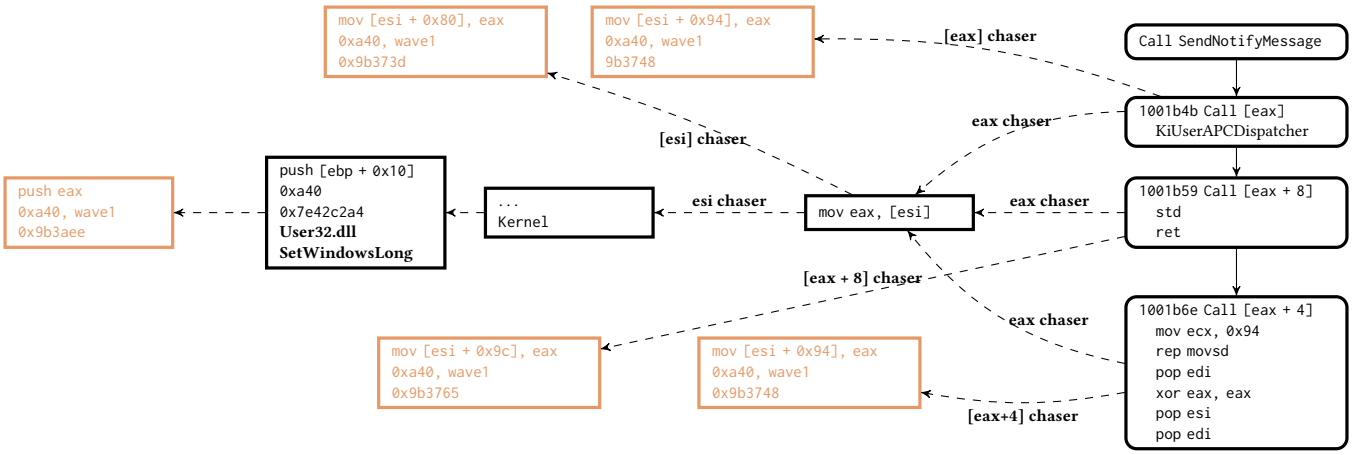


Figure 5: Code injection graph for the first three gadgets in Gapz code injection. The graph shows that in all three `call` gadgets the value tainting *eax* was propagated through `SetWindowsLong` in the host process.

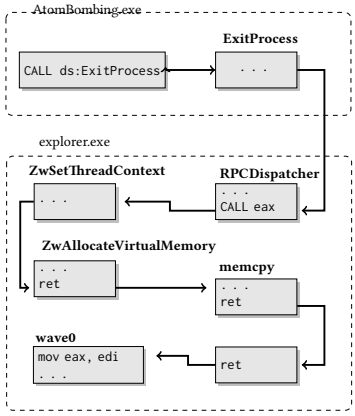


Figure 7: AtomBombing caught by Tartarus without any hooks.

is set to *ZwAllocateVirtualMemory* which is the first code-reuse attack in the target process, and *esp* is set 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 injecter process.

When Tartarus is matched with AtomBombing, Tartarus finds the code injection shown in Figure 6. Tartarus captures the code injection exactly, seeing that a call to `NtQueueApcThread` in the host process results in the target process calling `SetThreadContext`. The `ret` instruction inside `ZwAllocateVirtualMemory` is the first ROP gadget in the injection. This ROP gadget transfers execution to `memcpy` and the return instruction of `memcpy` transfers execution to a simple ROP gadget consisting of only a `ret` instruction. This gadgets 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 7, we show the AtomBombing injection caught by Tartarus when there are no hooks on `NtQueueApcThread`. In this case, Tartarus catches the exact same code injection except for the initiator instruction which in this case is a call to *ExitProcess*. The reason this happens is because *NtQueueAPCThread* is an asynchronous procedure call, which means that the last API call in the injector process at the time the injection happens inside *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 right initiator instruction and not to identify whether an injection has occurred or not.

7.4 Performance evaluation

The performance of Tartarus has a large impact on the applications of the tool. Because we specifically use Tartarus for malware analysis, we measure the performance of Tartarus relative to the malware samples in our data set. For a performance measurement specifically about DECAF, we refer to the original DECAF paper [21]. To put the performance of Tartarus in perspective, we measure how fast Tartarus identifies the instructions that are part of the system-wide CFG, i.e. unique instructions in the malware execution trace, during execution of the samples.

We selected one sample from each of the malware families in our B data set and ran them for 1100 seconds inside Tartarus. The samples shared very similar behaviours to the other samples in their respective families, so each sample represents well the overall malware family. Our experiments were performed on a laptop with an i7 Quad Core 2.5 GHz processor and 16 GB of ram and three instances of the dynamic analysis environment were run simultaneously, meaning 3 samples were analysed approximately every 1100 seconds on a standard laptop. We used a Windows XP SP3 image. Our experiments were performed with our taint-logging turned off, meaning Tartarus is able to identify code waves and code injections with their key components, but not the entire code injection graph. The results are shown in Figure 8.

We can see in Figure 8 that the majority of Tinba was captured in less than 2 minutes, Gapz about 13 minutes, Ramnit about 7 minutes

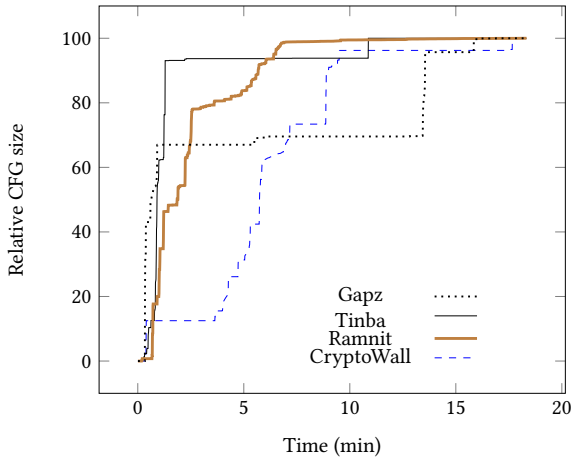


Figure 8: Size of CFG relative to analysis time.

	Total instructions	MT instructions	Last injection
Gapz	1.2B	225M	13Min
TinyBanker	230M	10M	2Min
CryptoWall	2.0B	250M	14Min
Ramnit	580M	50M	7Min

Table 4: Instruction count of malware samples. #MT = Malware execution trace.

and CryptoWall 4.0 in about about 17 minutes. To put this into perspective of the sample sizes, we ran the same experiment with two counters for capturing the number of instructions executed in entire system and the number of instructions in the malware execution trace. We counted until the last code injection in each of the samples. The instruction counts are shown in Table 4.

An interesting aspect of Figure 8 is how clearly it shows that the samples in our dataset work in stages in that the construction of the CFG is not linear but more of a step-wise construction. For example, the Gapz malware has revealed about 65% of its CFG within the first 3 minutes, but then continues for 10 with only about 5% more of the CFG constructed. After 13 minutes of execution time, Gapz finally reveals more of itself and a big leap to about 95% happens.

To put the construction of the CFG into perspective of dynamically generated code and code injections, Figure 9 and Figure 10 show the CFG construction relative to execution time, with the addition of markers for when the first wave of dynamically generated code and code injections occurred. The graphs are of a Tinba and a Gapz sample, respectively. Note the Tinba graph is zoomed in on the first 2 minutes of execution time. A triangle shows the first wave of dynamically generated code and a circle denotes a code injection. We observe that the majority of the code execution happens *after* the first wave of dynamically generated code. Namely, in the Tinba case, more than 90% of the malware execution happens after the first wave of dynamically generated code and in the Gapz sample about 60%. Furthermore, in the Tinba case more than 80% of unique instructions in the CFG happens after the first code injection, where in the Gapz sample more than 20% happens after the first code injection.

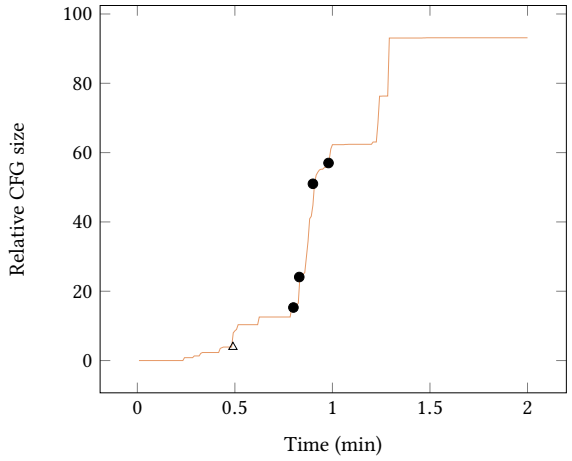


Figure 9: Relative CFG size of Tinba malware with first wave of dynamically generated code and code injections marked.

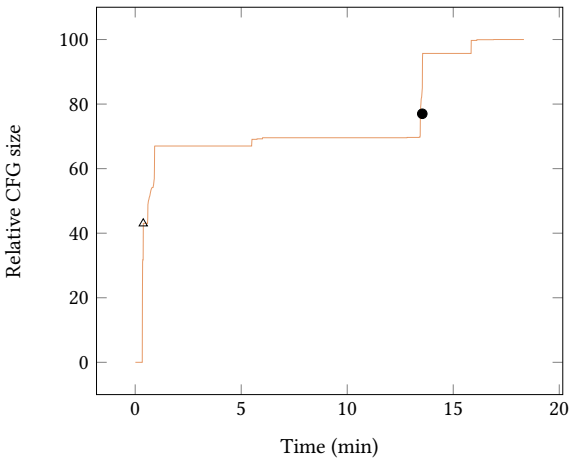


Figure 10: Relative CFG size of Gapz malware with first wave of dynamically generated code and code injections marked.

7.5 Relevance of approach on recent malware

We match Tartarus with a recent collection of malware samples to demonstrate the relevance of our approach. In total, we analyse 934 PE files submitted to VirusTotal in April and May of 2017. Each sample has at least 40 anti-malware vendors reporting the sample malicious. We verify the relevance of our approach by counting the number of processes a sample injects into and whether it uses code-reuse attacks. We execute each sample for 600 seconds.

We found that 373 samples inject code into other processes, which corresponds to 40% of the total malware samples. In contrast, a report from PaloAlto Networks on *New and Evasive Malware* from 2013 reports that code injection was observed in 13.5% of samples

collected in March 2013 [33]. This is an increase of almost three times in four years. We found that 223 of the samples inject into 4 processes or less and in 120 of the samples code-reuse attacks were used as part of a code injection. In 118 of these cases, only one code-reuse attack was used and in the other two cases a chain of two code-reuse attacks were used. We consider this to be a strong indication that indeed code-injection is increasingly being used, but chained code-reuse attacks inside code injections remain rare.

8 LIMITATIONS AND FUTURE WORK

In this work our focus is on tracing malware propagations within the host and identifying code injections and code waves with a special attention to the use of code-reuse attacks. In this section we describe limitations of our approach and some of the remaining challenges in the problem space of capturing and describing malware propagations.

An interesting limitation to our approach is that we only consider malware propagations that happens within a single system execution. However, some malware carry propagation strategies that stretch over a system reboot by, for example, dropping a bootkit or rootkit which is initialized during system start-up. In fact, the Gapz malware is in this category. When Gapz has injected into *explorer.exe* it continues to drop a bootkit and then modify either the master boot record (MBR) or the volume boot record (VBR) (depending on the version of Gapz). As a result, during the next bootup, the modifications to the MBR or VBR will cause the bootkit to be loaded and Gapz will continue execution via the bootkit. Automatic analysis of this part of Gapz requires dynamic analysis to be stretched over the system boot. To the best of our knowledge, dynamic analysis of malware has never been done over several system boots. Therefore, an interesting avenue of further research is to investigate how much more information can be leveraged automatically about the malware with such an approach.

Another limitation to our approach is the use of our techniques when the guest system is a multiprocessor environment. For example, our approach to collecting the malware execution trace described in Algorithm 1 assumes a single-core execution environment. Although we believe the identification of code injections and code waves can follow very similar strategies to our current approach, we have not yet tested Tartarus with a multi-core guest environment. However, we will have many more parallel execution contexts to concern ourselves with and can potentially not rely on the order of the malware execution trace as we do in this work.

In this paper we focused on identifying control-flow aspects between processes based on code injections and code waves within each process. However, we have not paid a lot of attention to the control-flow aspects within each code wave itself. In most cases, we can identify a lot of the control-flow within each code wave from the instruction execution order in the malware execution trace and disassembly of the code wave. However, if a malware sample writes memory to another process and this memory has several different entry points independent of each other, then our approach will only recognize one code injection and consider the written memory as one code wave. A more accurate approach would be to discover multiple code injections to the same code wave, and use that information to capture more precisely the control-flow within the code wave itself. One potential strategy for solving this problem is a more refined definition of a code-wave, for example something similar to Ugarte et al. who divides the memory of a code wave into *unpacking frames*. However, this would require us to relax the way we identify code injections and may end up producing many false positives in the control-flow graph of the malware.

From a practical point of view, a limitation to our approach is that malware can detect the use of dynamic analysis. This problem is shared by any dynamic analysis environment. For example, malware can detect the presence of QEMU and then diverge execution to non-malicious behaviours. We do not perform any activities to combat malware that tries to detect the presence of our system. There exists several approaches to hardening QEMU for malware analysis which are directly correlated with approaches in which malware detects QEMU. If we implement these techniques into Tartarus we may harden it for some time, but is not likely to work as a general solution. Conceptually, in order to make it harder for the malware to detect that it is running inside an analysis environment, one approach is to switch to dynamic analysis environment that are more transparent in nature. For example, environments like Ether [12] that utilize hardware virtualization extensions offer more separation between the analysis environment and the guest environment in which the malware is executing.

On a fundamental level, our techniques rely on taint analysis for capturing malware propagation. As such, our techniques inherit the limitations of taint analysis for malware analysis, meaning an attacker can deploy information-flow evasive behaviours in order to avoid analysis by our techniques. Certainly, techniques like multi-path exploration via symbolic execution can aid in defeating evasive behaviours. However, given that we can't explore the entire state-space of the vast majority of malware samples, the question we must really solve is *how to identify evasive behaviours*. When an evasive behaviour is then detected, we can rely on various techniques, with symbolic execution as one of them, to guide execution down the path of interest. For a more general description about limitation to taint analysis for malware analysis we refer to Cavallaro et al. [7].

9 RELATED WORK

Dynamic taint analysis have many applications in automating malware analysis tasks and a lot of work has been done in this area. System-wide fine-grained malware analysis with taint information was first proposed in Panorama by Yin et al. [46]. Panorama is built on top of QEMU and offers multiple features such as keylogger detection and malware tracing based on taint analysis. Indeed the malware tracing offered by Panorama is the work the comes closest to ours. In the malware execution tracing itself, Tartarus differs from Panorama by also considering code-reuse attacks and initially taints more memory than Panorama which only taints the text section. Panorama deploys no abstractions on the execution trace itself where Tartarus abstracts the trace into code waves and code injections that are then used to construct a system-wide CFG. Finally, the evaluation performed with Panorama is centred around its keylogger detection, which does not rely on malware tracing via taint, and the evaluation presented in this work is therefore the first evaluation on taint analysis for malware tracing. Other work include Egele et al. who use dynamic taint analysis for spyware

detection [13] and Moser et al. who use dynamic taint analysis in combination with linear constraint solvers to explore multiple execution paths in malware [32]. Dynamic taint analysis has also been explored in Android, most notably by TaintDroid [14] and DroidScope [44]. Besides automatic approaches to malware analysis, dynamic taint analysis has also been proposed as an aid to manual reverse engineering tasks. For example, the tool SemTrax [26], which is no longer being developed, augments debugging with dynamic taint analysis and has the ability to visualize various relationships on tainted data, such as the set of operations performed on tainted memory.

In the last decade, full-system dynamic binary analysis platforms have gained a lot of attention from malware analysis researchers. As a result, there now exists several multi-purpose systems that are well-suited for building further analysis tools. We designed Tartarus on top of DECAF [20] which is the successor of TEMU from the BitBlaze project [39]. One of the key features of DECAF is that taint analysis is performed directly on the QEMU tcg instructions, which allows it to perform fast taint analysis during execution. In addition to fast tainting via the tcg instructions, DECAF performs bitwise taint analysis which gives it a very high level of precision in its taint. PANDA is another platform that allows full-system dynamic binary analysis. In comparison to DECAF, PANDA is built around the concept of repeatable reverse engineering. It executes an instance of QEMU and *records* all non-deterministic data that goes into the system. With the non-deterministic data recorded, PANDA then provides the ability to *replay* the execution and the actual analysis of the execution is performed during *replay*. The key advantages of this strategy compared to performing analysis directly on the real execution is that the recording minimally affects the actual execution in terms of performance, and that analysis can take a stepwise fashion. As such, the execution on which analysis is performed is effectively independent of the computational workload of the analysis, i.e. analysis is performed on a "fast" full-system emulation, and, different analysis can be performed on the same execution. PANDA supports byte-level taint analysis by raising the QEMU trace into an LLVM trace. S2E [9] is another platform that also offers full-system dynamic binary analysis, but, in comparison to DECAF and PANDA, is focused around augmenting symbolic execution to full-system analysis. S2E does this via an x86-to-LLVM QEMU backend that interfaces S2E with the KLEE symbolic execution engine that interprets LLVM instructions. Both DECAF, PANDA and S2E provides comprehensible interfaces for writing plugins and are all open-source projects.

Throughout the paper we have already mentioned automated unpackers and other tools that give solutions to the problem of self-modifying code. We have specifically focused on Codisasm [5] and Ugarte et al [40]. Other tools include Renovo [22], OmniUnpack [30], EtherUnpack [12], RePEconstruct [24] and Polyunpack [37]. All of these tools rely on the heuristic of monitoring explicit write-then-execute patterns. EtherUnpack and Renovo captures dynamically generated code within a specific process where OmniUnpack deploys a more coarse-grained approach by monitoring for page-level write-then-execute patterns and suspicious system calls. RePEconstruct is a tool based on DynamoRIO that aims at uncovering dynamically generated code and identify obfuscated API calls, but does not make any attempt to follow the malware across processes. Polyunpack detects unpacking behaviour by single-stepping an application and matching the program counter to an initial static analysis.

Besides related work in the more general scope of malware analysis, memory forensics is an area that also provides solutions to the problem of identifying how malware infects a system. For example, the Gapz malware places shellcode within *explorer.exe* by overwriting the function *atan* inside of *ntdll*. Code integrity check to verify the provenance of code in memory images is able to identify that malware overwrote the function by analysing memory images [42]. Volatility [15] is a popular open source project that is used for digital forensics and indeed there are plugins for volatility aimed at identifying code injection [31]. The difference between our approach and that of memory forensics is the forensics analysis relies on a snapshot where our approach is based on analysis of 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, whereas we would still observe the execution of shellcode.

10 CONCLUSION

In this paper we concern ourselves with automatic analysis of host-based malware propagations. Specifically, we divide the problem into to two smaller tasks. First, how to trace malware in a general and precise manner in the context of execution across several processes and code-reuse attacks. Second, how to raise the collected execution trace into higher-level semantics of dynamically generated code and code injections.

To solve these problems we have proposed three techniques and implemented them in a system we call Tartarus. Tartarus is a malware analysis environment that traces malware execution based on taint analysis and a model of code-reuse attacks. Tartarus abstracts the execution trace into code waves based on an information-flow model, and also identifies and highlights intrinsic characteristics about code injection techniques in the execution trace. Finally, Tartarus combines these abstractions into a system-wide control-flow graph.

We test Tartarus in the context of ground-truth applications and our results show that Tartarus accurately captures malware propagations, even without prior knowledge about them. We show via a comparative evaluation that Tartarus improves capture of malware execution traces over state-of-the-art dynamic analysis tools. We evaluate the performance of Tartarus which ranges from a few minutes to 15 minutes depending on the complexity of the sample. Finally we demonstrate the relevance of our approach by matching Tartarus with a recent malware data set which shows the number of malware samples that inject code have increased almost three times since 2013.

ACKNOWLEDGMENTS

The authors would like to acknowledge our anonymous reviewers, Pedro Antonino and Julien Vanegue for useful feedback and insightful critique. We would also like to thank VirusTotal for providing

malware samples and Udi Yavo and Tal Liberman of enSilo for making their code injection techniques available. Finally we would like to thank Xunchao Hu for helping the first author with several aspects of DECAF. Work is funded by National Science Foundation Grant #1664315 and DARPA Grant #FA8750-16-C-0044.

REFERENCES

[1] Andrea Allievi and Holger Unterbrink. 2015. CryptoWall 4 The Evolution Continues. (2015).

[2] Magal Baz and Or Safran. 2017. Dridex's Cold War: Enter AtomBombing. (2017).

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[4] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. 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)*. ACM, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[5] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. 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)*. ACM, New York, NY, USA, 745–756. https://doi.org/10.1145/2810103.2813627

[6] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. 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)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/1455770.1455776

[7] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2008. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08)*. Springer-Verlag, Berlin, Heidelberg, 143–163. https://doi.org/10.1007/978-3-540-70542-0_8

[8] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. 2009. 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)*. USENIX Association, Berkeley, CA, USA, 6–6.

[9] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Trans. Comput. Syst.* 30, 1, Article 2 (Feb. 2012), 49 pages. https://doi.org/10.1145/2110356.2110358

[10] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. 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)*. USENIX Association, Berkeley, CA, USA, 401–416. http://dl.acm.org/citation.cfm?id=2671225.2671251

[11] Cuckoo developers. 2017. Cuckoo Sandbox. (2017). https://www.cuckoosandbox.org/

[12] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1455770.1455779

[13] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 18, 14 pages. http://dl.acm.org/citation.cfm?id=1364385.1364403

[14] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. 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)*. USENIX Association, Berkeley, CA, USA, 393–407. http://dl.acm.org/citation.cfm?id=1924943.1924971

[15] Volatility Foundation. Volatility - Open Source Memory Forensics. (????). http://www.volatilityfoundation.org/

[16] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 417–432. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas

[17] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. 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)*. ACM, New York, NY, USA, 47–58. https://doi.org/10.1145/2897845.2897894

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

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

[20] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 248–258. https://doi.org/10.1145/2610384.2610407

[21] Andrew Henderson, Lok Kwong Yan, Xunchao Hu, Aravind Prakash, Heng Yin, and Stephen McCamant. 2017. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. *IEEE Trans. Softw. Eng.* 43, 2 (Feb. 2017), 164–184. https://doi.org/10.1109/TSE.2016.2589242

[22] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM '07)*. ACM, New York, NY, USA, 46–53. https://doi.org/10.1145/1314389.1314399

[23] Thomas Kittel, Sebastian Vogl, Julian Kirsch, and Claudia Eckert. 2015. *Counteracting Data-Only Malware with Code Pointer Examination*. Springer International Publishing, Cham, 177–197. https://doi.org/10.1007/978-3-319-26362-5_9

[24] D. Korczynski. 2016. RePEconstruct: reconstructing binaries with self-modifying code and import address table destruction. In *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. 1–8. https://doi.org/10.1109/MALWARE.2016.7888727

[25] Peter Kruse. 2012. W32.Tinba (TinyBanker) The Turkish Incident. (2012).

[26] Persistence Labs. 2013. Semtrax. (2013). http://www.persistencelabs.com/blog

[27] Tal Liberman. 2016. AtomBombing: Brand New Code Injection for Windows. (2016).

[28] Tal Liberman. 2017. BSidesSF 2017, AtomBombing: Injecting Code Using Windows' Atoms. (2017). https://www.youtube.com/watch?v=9HV69QGiBAU

[29] Wayne Low. 2012. Code injection via return-oriented programming. *Virus Bulletin* (2012).

[30] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *In Proceedings of the Annual Computer Security Applications Conference (ACSAC*.

[31] Monnappa22. HollowFind. (????). https://github.com/monnappa22/HollowFind

[32] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP '07)*. IEEE Computer Society, Washington, DC, USA, 231–245. https://doi.org/10.1109/SP.2007.17

[33] PaloAlto Networks. 2013. The Modern Malware Review. (2013).

[34] Michalis Polychronakis and Angelos D. Keromytis. 2011. ROP Payload Detection Using Speculative Code Execution. In *Proceedings of the 2011 6th International Conference on Malicious and Unwanted Software (MALWARE '11)*. IEEE Computer Society, Washington, DC, USA, 58–65. https://doi.org/10.1109/MALWARE.2011.6112327

[35] Symantec Security Response. 2015. W32.Ramnit analysis. (2015).

[36] Eugene Rodionov and Aleksandr Matrosov. 2016. Mind the Gapz: The Most Complex Bootkiv Ever Analyzed? (2016).

[37] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of the 22Nd Annual Computer Security Applications Conference (ACSAC '06)*. IEEE Computer Society, Washington, DC, USA, 289–300. https://doi.org/10.1109/ACSAC.2006.38

[38] Hovav Shacham. 2007. 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)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[39] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[40] 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. (????).

[41] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. 2014. Persistent Data-only Malware: Function Hooks without Code. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS)*.

[42] Andrew White, Bradley Schatz, and Ernest Foo. 2013. Integrity verification of user space code. *Digital Investigation* (2013).
[43] Lok Yan and Heng Yin. 2017. SoK: On the Soundness and Precision of Dynamic Taint Analysis. (2017). http://www.cs.ucr.edu/~heng/teaching/cs260-winter2017/formaltaint.pdf
[44] Lok Kwong Yan and Heng Yin. 2012. 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)*. USENIX Association, Berkeley, CA, USA, 29–29. http://dl.acm.org/citation.cfm?id=2362793.2362822
[45] Udi Yavo and Tomer Bitton. 2015. Injection on Steroids: Code-less Code Injections and 0-Day Techniques. (2015).
[46] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. 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)*. ACM, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

A FORMAL COMPARISON TO PREVIOUS WORK

The definition of malware execution trace given in Definition 1 allows us to formally compare Tartarus to previous works [5, 40]. Before we make the comparison, however, we need to give a few more definitions. First, for a given instruction, let $i[W]$ denote the memory it writes, $i[A]$ the instruction address and $i[O]$ the output of the instruction. We have that $i[W] \subseteq i[O]$ and for convenience that $i[A] \subseteq i[W] \subseteq i[O] \subseteq S$. Second, let the function $TP : S \times i \rightarrow S$ represent the taint propagation function, and $CR : S \times i \rightarrow \{true, false\}$ be the predicate that identifies code-reuse.

We now proceed to formally define the malware analysers of previous work [5, 40] and the approach by Tartarus, and then continue to show malware execution trace collected by previous work is a subtrace of the one collected by Tartarus. Both [5, 40] propose the malware analyser $P_{A,UB}$ that updates the shadow memory with any memory written by instructions already in the shadow memory, as such they define δ_A as follows:

$$\delta_{A,UB}(s,i) = s \cup t_{UB} \text{ where } t_{UB} = \begin{cases} i[W], & \text{if } i[A] \in s \\ \varnothing, & \text{otherwise} \end{cases}$$

Furthermore, in the malware execution trace they only include instructions that are part of the shadow memory, and therefore defined Λ_A as follows:

$$\Lambda_{A,UB}(s,i) = \begin{cases} \text{true}, & \text{if } i[A] \in s \\ false, & \text{otherwise} \end{cases}$$

Tartarus updates the shadow memory based on the taint propagation function TP and also all output by instructions in the shadow memory. As such, the malware analyser Tartarus $P_{A,TA}$ defines δ_A as follows

$$\delta_{A,TA}(s,i) = s \cup t_{TA} \text{ where } t_{TA} = \begin{cases} TP(i,s) \cup i[O] & \text{if } i[A] \in s \\ TP(i,s) & \text{otherwise} \end{cases}$$

Furthermore, Tartarus includes in the malware execution trace any instruction part of the shadow memory or part of a code-reuse attack, given by the CR function, and therefore defines Λ_A as follows

$$\Lambda_{A,TA}(s,i) = \begin{cases} \text{true}, & \text{if } i[A] \in s \vee CR(i,s) \\ false, & \text{otherwise} \end{cases}$$

We now show shat the malware execution trace collected by previous work is indeed a subset of the malware execution trace collected by Tartarus.

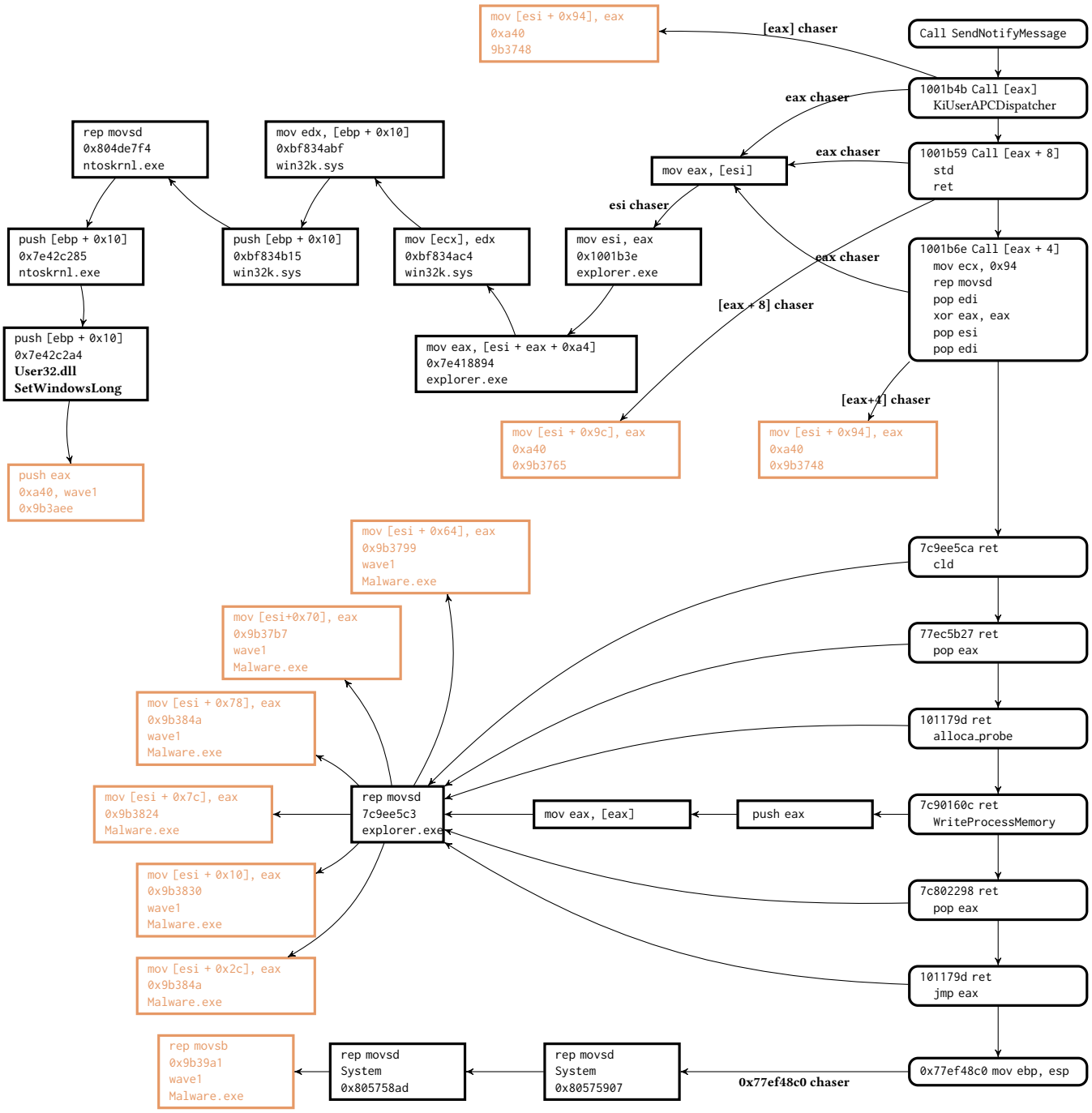
Theorem A.1. *Let $P_{A,UB}$ and $P_{A,TA}$ be the malware tracer defined above and $T(P,E)$ be some execution trace. Furthermore, let $ST_{UB}(T(P,E)) = (s_{u,0}, \ldots s_{u,l})$ and $ST_{TA}(T(P,E)) = (s_{t,0}, \ldots, s_{t,l})$ be the respective shadow memory sets. Given that $s_{u,0} = s_{t,0}$, we have that there is no element $i \in T(P,E)$ such that $i \in \Pi_{UB} \wedge i \notin \Pi_{TA}$.*

Proof. We prove by contradiction that there is no element $i \in T(P,E)$ such that $i \in \Pi_{UB} \wedge i \notin \Pi_{TA}$.

We first show that each element in the shadow memories collected by $P_{A,UB}$ is a subset of the corresponding element in the shadow memories collected by $P_{A,TA}$. We have that $s_{u,0} = s_{t,0}$ and therefore $s_{u,0} \subseteq s_{t,0}$. From the definitions of $\delta_{A,UB}$ and $\delta_{A,TA}$, we then have have $t_{UB} \subseteq t_{TA}$ and therefore $\delta_{A,UB}(s_{u,0}, i_0) \subseteq \delta_{A,TA}(s_{t,0}, i_0)$. Because δ_A is a monotonic increasing function, we then have that $\forall s_{u,i} \in ST_{UB}(T(P,E)) | s_{u,i} \subseteq s_{t,i}$. As such, each element in the shadow memories collected by $P_{A,UB}$ is a subset of the corresponding element in the shadow memories collected by $P_{A,TA}$.

Assume there exist an element m such that $m \in \Pi_{A,UB} \wedge m \notin \Pi_{A,TA}$. This means there exists an element $i \in T(P,E)$ such that $\Lambda_{A,UB}(s_{u,j}, i)$ is true and $\Lambda_{A,TA}(s_{t,j}, i)$ is false. From the definitions of $\Lambda_{A,UB}$ and $\Lambda_{A,TA}$, this is only possible if $s_{u,j}$ contains an element that is not in $s_{t,j}$. We have shown above that this is not the case, and the proof is done. □

B CODE INJECTION GRAPH OF GAPZ SAMPLE.


mov [esi + 0x94], eax
0xa40
9b3748
[eax] chaser
Call SendNotifyMessage
1001b4b Call [eax]
KiUserAPCDispatcher
eax chaser
rep movsd
0x804de7f4
ntoskrnl.exe
mov edx, [ebp + 0x10]
0xbf834abf
win32k.sys
mov eax, [esi]
eax chaser
1001b59 Call [eax + 8]
std
ret
esi chaser
push [ebp + 0x10]
0x7e42c285
ntoskrnl.exe
push [ebp + 0x10]
0xbf834b15
win32k.sys
mov [ecx], edx
0xbf834ac4
win32k.sys
mov esi, eax
0x1001b3e
explorer.exe
eax chaser
1001b6e Call [eax + 4]
mov ecx, 0x94
rep movsd
pop edi
xor eax, eax
pop esi
pop edi
[eax + 8] chaser
push [ebp + 0x10]
0x7e42c2a4
User32.dll
SetWindowsLong
mov eax, [esi + eax + 0xa4]
0x7e418894
explorer.exe
[eax+4] chaser
mov [esi + 0x9c], eax
0xa40
0x9b3765
mov [esi + 0x94], eax
0xa40
0x9b3748
push eax
0xa40, wave1
0x9b3aee
mov [esi + 0x64], eax
0x9b3799
wave1
Malware.exe
7c9ee5ca ret
cld
mov [esi+0x70], eax
0x9b37b7
wave1
Malware.exe
77ec5b27 ret
pop eax
mov [esi + 0x78], eax
0x9b384a
wave1
Malware.exe
101179d ret
alloca_probe
mov [esi + 0x7c], eax
0x9b3824
Malware.exe
rep movsd
7c9ee5c3
explorer.exe
mov eax, [eax]
push eax
7c90160c ret
WriteProcessMemory
mov [esi + 0x10], eax
0x9b3830
wave1
Malware.exe
7c802298 ret
pop eax
mov [esi + 0x2c], eax
0x9b384a
Malware.exe
101179d ret
jmp eax
rep movsb
0x9b39a1
wave1
Malware.exe
rep movsd
System
0x805758ad
rep movsd
System
0x80575907
0x77ef48c0 chaser
0x77ef48c0 mov ebp, esp