

MASTER THESIS
COMPUTING SCIENCE: CYBER SECURITY



RADBOD UNIVERSITY

Detecting Capabilities in Malware Binaries by Searching for Function Calls

Author:
Joren Vrancken
s4593847

Supervisor & First Assessor:
dr. ir. Erik Poll
erikpoll@cs.ru.nl

Second assessor:
dr. Freek Verbeek
freek@vt.edu

June 15, 2022

Acknowledgments

I would like to sincerely thank everybody that helped me write this thesis. In particular, I would like to thank Erik Poll. His guidance and thoughtful questions about my approach made me think deeper about the research. It was a joy and great learning experience to work with him. Multiple people have provided feedback and guidance throughout the research and proofread the thesis. My wholehearted thanks to you all. This thesis has become much better because all of you.

Abstract

Incidents like the ransomware attack on the Colonial Pipeline in 2021 [37] show that malware is a growing threat with real-world implications. To combat this growing threat, there is an ever-growing need for (automated) malware analysis tools.

Meaningful actions by applications (including malware) require interaction with the operating system. Operating systems facilitate this by providing an API. Malicious activity is often implemented via calls to this API. Such calls can be seen as an indicator of the malicious activity.

As most malware is written for Windows [35], we focus on Windows and the Windows API.

In this thesis, we present a method to detect malware capabilities by searching for specific function calls with specific arguments. This method consists of two steps: (1) describing the function calls that are used to implement a capability as patterns, including the arguments that are passed to such functions, and (2) searching for these patterns in malware binaries.

For the first step, we provide patterns to describe function calls: *Call Signatures*. For the second step, we developed an IDA Pro plugin, called Call Signatures Plugin (*CSP*), that uses Call Signatures to search for function calls in a binary. This plugin is available at <https://github.com/joren485/CallSignaturesPlugin>.

To showcase the effectiveness of detecting capabilities with Call Signatures and CSP, we perform an experiment where we use them to detect a capability that is commonly implemented by malware: *persistence*. Persistence is the class of techniques that malware use to maintain access to a victim system after an initial infection. We provide an overview of common persistence techniques and the function calls that are required to implement these techniques. We express these function calls, including their arguments, as Call Signatures and then use CSP to search for these techniques in a dataset of real-world malware samples.

The outcome of this experiment shows that function calls can be an effective indicator for detecting malware capabilities. We also show that Call Signatures and CSP perform well compared to existing tooling.

Contents

1	Introduction	4
2	Related Work	7
3	Background on x86 & Windows	9
3.1	The x86 Architecture	9
3.1.1	Registers	10
3.1.2	The Stack	11
3.1.3	Assembly	12
3.2	Function Calls	15
3.2.1	Calling Conventions	16
3.2.2	Decompiling Function Calls	18
3.3	Windows	21
3.3.1	The Windows API	21
3.3.2	The Windows Registry	24
3.3.3	Scheduled Tasks & Services	26
4	Background on Malware & Malware Analysis	29
4.1	Malware Analysis	30
4.1.1	The Advantages & Limitations of Static Analysis . . .	31
4.1.2	IDA Pro	31
5	Call Signatures	33
5.1	The Syntax of Call Signatures	34
5.1.1	Rules	36
5.2	Examples of Call Signatures	39
6	Call Signatures Plugin	42
6.1	Call Signature Matching	44
6.2	Examples of Call Signatures Matching	44
7	Persistence Techniques	47
7.1	Registry-based Persistence Techniques	48
7.2	Directory-based Persistence Techniques	50

7.3	Service-based Persistence Techniques	50
7.4	Scheduled Task-based Persistence Techniques	51
8	Call Signatures for Persistence Techniques	52
8.1	Registry-based Persistence Techniques	52
8.1.1	Modifying the Registry via the API	53
8.1.2	Modifying the Registry via Reg.exe	53
8.2	Directory-based Persistence Techniques	54
8.2.1	Using the Path of the Startup Directory Directly	55
8.2.2	Resolving the Path of the Startup Directory	55
8.3	Service-based Persistence Techniques	58
8.3.1	Creating Services Using the Windows API	59
8.3.2	Creating Services Using Sc.exe	59
8.3.3	Creating Services Using the Registry	60
8.4	Scheduled Tasks-based Persistence Techniques	60
8.4.1	Creating Services Using the Windows API	60
8.4.2	Creating Services Using Schtasks.exe	61
8.4.3	Creating Services Using At.exe	62
8.5	A Reflection on Writing Call Signatures	63
9	Experiments & Analysis	65
9.1	Detection of Persistence Techniques in Real-World Samples	65
9.1.1	The Sources of Malware Samples	66
9.1.2	Creating the Datasets	66
9.1.3	Analyzing the Datasets	67
9.1.4	Discussion of the Results	67
9.2	Comparison With Capa	69
9.2.1	A Short Overview of Capa	69
9.2.2	The Differences Between Call Signatures & Capa Rules	70
9.2.3	Experiments	71
9.2.4	False Positives	73
10	Future Work	75
11	Conclusions	77
	Bibliography	79
A	Finding Common Traits Using Binlex	83
B	Example Code Used in the Dynamic Linking Comparison	86

C	Dataset Hashes	89
C.1	The Hashes of Dataset RP	89
C.2	The Hashes of Dataset DP	93
C.3	The Hashes of Dataset SP	97
C.4	The Hashes of Dataset TP	100
C.5	The Hashes of the Binx Dataset	101

Chapter 1

Introduction

Malware is software that is made with malicious intent. Its goal is to disrupt or damage computer systems, users, or networks [22] [34]. Criminals have started using malware for monetary gain. For example, by stealing banking and credit card information [8] or by disrupting systems until a ransom has been paid [30]. The global intelligence community has also embraced malware to stealthily infiltrate the systems of adversaries [31]. Their goals are not monetary¹, but rather stealing sensitive information, espionage, or disruption of vital sectors [41] [11].

The increased use of malware has increased the need for defense against and analysis of malware. Today many companies provide commercial antivirus suites that will scan consumer systems for known malware and malware behavior (e.g. Norton², Kaspersky³, and F-Secure⁴). Other companies focus on helping larger organizations protect themselves and perform incident response once a breach has been detected (e.g. Mandiant⁵ and CrowdStrike⁶). This has created a digital arms race where malware authors will use evermore advanced techniques to evade detection and malware analysts will use evermore advanced techniques to detect malware.

Malware analysis is the process of studying malware samples to try to determine their functionality, origin, and targets [34]. It plays an important role in the defense against malware. Malware analysis helps understand known malware and discover unknown malware.

According to the AV-test institute, about four hundred fifty thousand new malware samples are found every day [2]. To process and analyze such a vast amount of data, we need tooling that automates malware analysis.

In this thesis, we look at how to statically detect high-level functionality

¹A notable exception to this is North Korea [5].

²<https://us.norton.com/>

³<https://www.kaspersky.com/>

⁴<https://www.f-secure.com/>

⁵<https://www.mandiant.com/>

⁶<https://www.crowdstrike.com/>

in malware binaries, by searching for patterns of function calls. Because API function calls are crucial to applications for any meaningful interaction with the operating system, we decided to detect high-level functionality using function calls (with specific arguments) as an indicator. Calls to API functions are harder to obfuscate than other parts of malware, as API functions are pre-defined by the operating system.

We define high-level functionality (i.e. capabilities) as *what* a malware sample is capable of (e.g. lateral movement, privilege escalation, and persistence). In contrast to low-level functionality, which describes *how* a malware sample implements the high-level functionality (e.g. implementing disk encryption with AES encryption). We choose to focus on high-level functionality as it is more helpful to a malware analyst to know what a sample is capable of. For example, it is generally more interesting to know that a sample encrypts files than that a sample uses AES encryption.

As most malware is written for Windows [35], we will focus on Windows binaries. Almost all newly sold Windows computers run a 64-bit version of Windows [12]. However, as 64-bit Windows provides backward compatibility for 32-bit executables [25], most malware is still written for 32-bit [12], as it allows a single executable to infect both 32-bit and 64-bit machines. Therefore, we will focus specifically on malware made for 32-bit Windows.

Before we discuss the research itself, we first go over related research (chapter 2) and background information. In chapter 3, we discuss background information on x86, function calls and Windows, and in chapter 4 we discuss malware and malware analysis.

We split the detection of high-level functionality into two steps:

1. Defining patterns for the function calls that are commonly used to implement the functionality. In chapter 5, we introduce such patterns: *Call Signatures*.
2. Searching for function calls that match these patterns in binaries. In chapter 6, we develop an IDA Pro plugin, called *CSP*, that uses Call Signatures to search for function calls in binaries. We choose to develop a plugin for IDA Pro, because IDA Pro is the most popular reverse engineering toolkit [7].

We test and showcase the usage of Call Signatures and CSP by using them to detect specific high-level functionality in real-world malware samples.

The high-level functionality we will focus on is persistence, the capability that describes how malware maintains access to an infected victim. We choose to focus on persistence as it is commonly implemented by malware [34].

First, we analyze which techniques are commonly used by malware to achieve persistence (in [chapter 7](#)). Second, in [chapter 8](#), we analyze the function calls used to implement these techniques and write Call Signatures to describe them.

In [chapter 9](#), we create datasets of malware samples that implement four different persistence techniques and run CSP on each dataset to see how well it can detect the persistence techniques. We also run another static analysis tool (Capa by Mandiant) on the datasets, to see how Call Signatures and CSP compare to existing tooling.

We conclude by looking at interesting ideas to expand upon this research ([chapter 10](#)) and with a discussion ([chapter 11](#)).

Chapter 2

Related Work

Analyzing the functionality of binaries (without running them) is an active field of research with multiple approaches. An important driver for analyzing the functionality of binaries is malware analysis, as malware authors are actively trying to circumvent existing analysis techniques.

A common way to detect and classify malware functionality is to look for code reuse between a potential malware sample and known malware samples. Because the source code of malware binaries is generally not available, assessing code reuse requires a similarity analysis of the binaries.

Haq and Caballero survey research on the similarity of binary code [13]. The research they survey focuses on semantic similarity, considering two binaries equivalent if they provide the same exact functionality. For example, a call graph-based approach by Lee et al. [18] and BinSim by Ming et al. [26] uses symbolic execution to analyze the logic in a binary. As there are countless ways to implement functionality, checking for semantic similarity is a complex process. To make matters worse, malware often implements obfuscation techniques to hide its functionality.

Other code reuse research looks at binaries as sequences of bytes. The general hypothesis being that similar source code results in similar compiled byte sequences. A popular technique is fuzzy-hashing: computing multiple hashes on sections of two inputs to find similar sections in the inputs [16]. For example, the research by Naik et al. [27] and Namanya et al. [28]. In [Appendix A](#) we see a weakness of such approaches: similar code can be compiled into vastly different binary code.

Machine learning is also being used to detect malicious functionality. As executable binaries are very complex data structures, either the neural network that is used to analyze the binaries needs to understand the general structure of such binaries or, pre-processing needs to take place to extract useful features to use in a neural network. For example, Raff et al. [29] and Massarelli et al. [21] opt for the former approach and Xu et al. [40] for the latter.

Many reverse engineering tools implement a database of commonly inlined functions (e.g. standard library functions). The most well-known being F.L.I.R.T. (Fast Library Identification and Recognition Technology) in IDA Pro [14]. Other examples are Function ID in Ghidra [1] and Talos FIRST (Function Identification and Recover Signature Tool) [39]. The goal of these databases is to find known (uninteresting) functions. This tells malware analysts, for example, which functions to skip when they are reverse engineering a malware sample. They work by creating signatures of commonly inlined functions (often the first 32 bytes of the function). Functions in a binary can be checked against these signatures.

The methodology we present in chapter 5 uses patterns to search for elements of interest (i.e. function calls) in a binary. In recent years there have been similar approaches. For example, FindFunc¹, a tool that allows for advanced searching of byte patterns in binaries.

Pattern matching is also used in source code analysis. For example, software that searches for common vulnerabilities in code (e.g. Semgrep², CodeQL³, and Weggli⁴) define patterns that describe these vulnerabilities to search for in the code.

Most similar to our research is Capa by Mandiant⁵. It identifies a broad spectrum of capabilities (e.g. cryptography, persistence, communication, and anti-analysis) in malware binaries. They define capabilities as rules of strings, byte sequences, and operating system API calls (similar to Yara signatures) that need to be present in functions. If all elements of a rule are present in a function, the sample is considered to have the capability that the rule defines. Unlike our research, their approach does not define a relation between the different elements that need to be present, which may cause false positives. We compare our work with Capa in chapter 9.

¹<https://github.com/FelixBer/FindFunc>

²<https://semgrep.dev/>

³<https://codeql.github.com/>

⁴<https://github.com/googleprojectzero/weggli/>

⁵<https://github.com/mandiant/capa>

Chapter 3

Background on x86 & Windows

We will first go over the necessary background information. In [section 3.1](#), we give general information on the CPU architecture that is used by Windows, x86. As this provides information on general concepts (registers, memory management, and assembly), it can be skipped by people that are familiar with these subjects. After that, we will discuss function calls and how they work in assembly, in [section 3.2](#). Finally, we will discuss Windows. As Windows is a large, complex platform and operating system, we will focus on concepts that are used in this thesis: the Windows API, the Windows Registry, Windows Services, and Scheduled Tasks.

In the next chapter we will discuss malware ([chapter 4](#)).

3.1 The x86 Architecture

CPUs perform operations by executing instructions. The model that describes these instructions is called an *instruction set architecture*. x86, an instruction set architecture developed by Intel, is the most widely used instruction set architecture [\[33\]](#).

An instruction indicates what operation the CPU should perform on data, stored in *registers* ([subsection 3.1.1](#)) and in the computers' memory.

An important feature of an instruction set architecture is the size of the registers and memory addresses. For example, a 32-bit architecture means that the registers and memory addresses consist of 32 bits and can store 2^{32} different values. The 32-bit variant of the x86 architecture is called IA-32 (Intel Architecture, 32-bit). However, most modern systems use the 64-bit variant, x86-64. x86-64 has full backward compatibility with IA-32, meaning that applications compiled for IA-32 can run on x86-64 (as long as the operating system also supports running IA-32 binaries).

3.1.1 Registers

Registers are small storage units that a CPU uses to store data during computations. Registers are implemented on the CPU itself, making data access significantly faster than accessing data stored in memory. There are two types of registers: general purpose registers and special purpose registers.

General-purpose Registers

General-purpose registers are used to store data or a memory address (i.e. a pointer) to data. x86 (specifically IA-32) has eight general-purpose registers, each having a traditional purpose. However, most can also be used for other purposes.

- **eax** (extended¹ accumulator register): Used in arithmetic operations.
- **ebx** (extended base register): Used as a pointer to data stored in memory.
- **ecx** (extended counter register): Stores the counter in looping operations.
- **edx** (extended data register): Used in I/O operations.
- **esi** (extended source index): Stores the address of the input data in certain operations on strings.
- **edi** (extended destination index): Stores the address of the output data in certain operations on strings.
- **ebp** (extended base pointer): Stores the address of the base of the current stack frame.
- **esp** (extended base stack pointer): Stores the address to the top of the stack.

In IA-32 these registers are larger versions of the registers used in the 16-bit and 8-bit variants of x86. For backward compatibility, x86 allows access to the 16-bit and 8-bit registers as the lower half of the 32-bit registers. Similarly, the 32-bit registers can be accessed in x86-64. [Table 3.1](#) shows how these register sizes relate to each other, using the **eax** register as an example.

¹32-bit registers are called “extended” to distinguish them from their 16-bit counterpart. For example, the 32-bit **eax** register and the 16-bit **ax** register.

64-bit	rax		
32-bit		eax	
16-bit		ax	
8-bit		ah	al

Table 3.1: The relation between the different accumulator registers in variants of x86.

Special Purpose Registers

Besides the general-purpose registers, x86 also has registers that store specific information about the program state and the CPU state. Two important registers with a specific purpose are:

- **eflags** (extended flags): Stores the data of previous instructions (e.g. the result of a comparison of two values) and the processor state as booleans.
- **eip** (extended instruction pointer): Stores the address of the next instruction.

3.1.2 The Stack

A *stack* is a data structure to store elements. We can add elements to the stack and take elements from the stack, with the restriction that the element that was added last is always the first element that is taken from the stack. This makes a stack a LIFO (last in, first out) data structure. A stack has two operations:

- **Push**: Add a new element on top of the stack.
- **Pop**: Remove the top (i.e. the most recently added) element from the stack.

x86 uses a stack in memory to store the state of function calls, called the *call stack*. Each time a function is called, a *stack frame* is pushed to the call stack. A stack frame stores the arguments, return address, and local variables of a function. When a function returns, the stack frame is popped from the stack.

The call stack grows downwards. When an application starts, the stack pointer is set to some address. When data is pushed on the stack, the stack pointer is *decreased*. Likewise, when data is popped from the stack, the stack pointer is *increased*.

Figure 3.1 illustrates what the stack would look like if a function `f0` has called another function `f1`. As we can see, the call frame of the callee is at the top of the stack, on top of the stack frame of the caller.

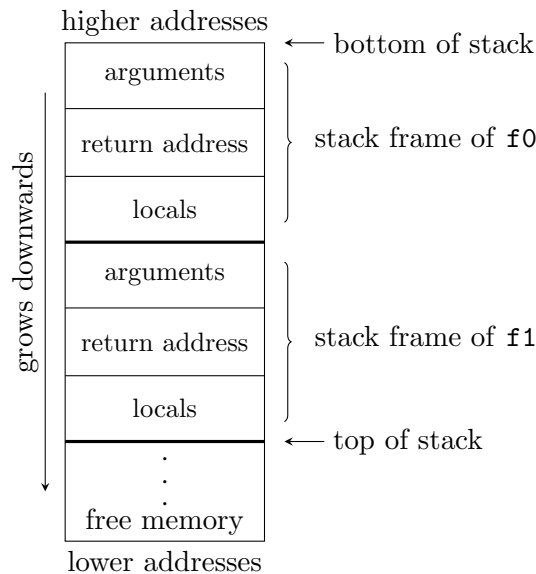


Figure 3.1: The layout of the stack when `f0` has called `f1`.

Not all data of a program is stored on the stack. Larger and dynamically allocated data is stored in the *heap*. Static and global variables are stored in a separate data segment.

3.1.3 Assembly

Assembly is a low-level programming language in which, generally speaking, each statement corresponds to one instruction executed by the CPU. Assembly statements are compiled into byte sequences of machine code called *opcodes*.

An assembly statement starts with the name of the instruction (the *mnemonic*), followed by its operands (i.e. arguments). [Listing 3.1](#) shows an assembly statement where `mov` is the mnemonic and `eax` and `1` are the operands.

```
1 mov eax, 1
```

Listing 3.1: A single instruction moving the integer `1` into the register `eax`.

x86 assembly code is written in either Intel syntax (mostly used for Windows development) or AT&T syntax (mostly used for Linux development). As we focus on Windows in this thesis, we will use the Intel syntax style.

In this section, we will discuss some basic instructions. There are, however, many more instructions used to efficiently perform operations on specific data types.

Arithmetic Instructions

x86 provides instructions for basic arithmetic.

- `add op0, op1`: Add the value of `op1` to `op0`.
- `sub op0, op1`: Subtract the value of `op1` from `op0`.
- `imul op0, op1`: Multiply the value of `op0` with `op1` and store the result in `op0`.
- `idiv op0`: Divide the contents of `edx:eax` (a 64-bit value of which the 32 most significant bits are taken from `edx` and the 32 least significant bits are taken from `eax`) by `op0` and store the result in `eax`.
- `inc op0`: Increase the value of `op0` by 1.
- `dec op0`: Decrease the value of `op0` by 1.

Logical Instructions

x86 provides instructions for logical operations.

- `and op0, op1`: Compute the bitwise and of `op0` and `op1` and store the result in `op0`.
- `or op0, op1`: Compute the bitwise or of `op0` and `op1` and store the result in `op0`.
- `xor op0, op1`: Compute the bitwise exclusive or of `op0` and `op1` and store the result in `op0`.
- `not op0`: Compute the bitwise not of `op0` and store the result in `op0`.

Data Movement Instructions

x86 provides instructions for moving data between memory and registers.

- `mov op0, op1`: Move the data stored in `op1` into `op0`.
- `lea op0, op1`: Move the address (i.e. pointer) stored in `op1` into `op0`.

Stack Instructions

x86 provides instructions for manipulating the call stack.

- `push op0`: Push `op0` to the stack. This decreases the stack pointer. As pushing data to the stack is essentially writing the data to memory, `push eax` is equivalent to [Listing 3.2](#).


```

1 sub esp, 4
2 mov [esp], eax

```

Listing 3.2: The `push eax` instruction written in terms of `sub` and `mov`.

- `pop op0`: Pop the top element from the stack and store it in `op0`. This increases the stack pointer. `pop eax` is equivalent to Listing 3.3.

```

1 mov eax, [esp]
2 add esp, 4

```

Listing 3.3: The `pop eax` instruction written in terms of `mov` and `add`.

Control Flow Instructions

x86 also provides instructions to control the flow of a program. This allows for subroutines (i.e. functions) to be defined and called. It also allows for conditional branching.

- `call op0`: Call a subroutine defined at `op0`.
A `call` performs two operations:
 1. It pushes the address after the `call` instruction to the stack (This is the *return address*).
 2. It changes the `eip` to the address that is being called (i.e. the next instruction will be at the address that is being called).

If `op0` is an address, the call is *direct*, because the function that is being called is known at compile time or load time. If `op0` is a register, the call is *indirect* as the address to the function that is called is determined at runtime.
- `ret`: Signal the end of a subroutine and return to the caller. It pops the return address from the stack and jumps to it.
- `jmp op0`: Jump to the instruction at `op0`.
- `cmp op0, op1` and conditional jumps: It is possible to only jump if a condition is met. The `cmp` instruction compares the values of its two operands and stores the result in the special `eflags` register. A conditional jump instruction (e.g. `je op0`, `jne op0` and `jge op0`) reads this result and jumps if its specific condition is met. For example, `je op0` jumps to `op0` when the operands of `cmp` are equal.

Because of the control-flow instructions, assembly code is not sequential, but rather a graph structure that can loop and skip code. The code between two control flow instructions is run sequentially and called a *basic block*. This creates a *control flow graph* of basic blocks and paths between these blocks. The conditionals in the code decide which paths are taken. Figure 3.2 shows an example of the control flow graph of a function.

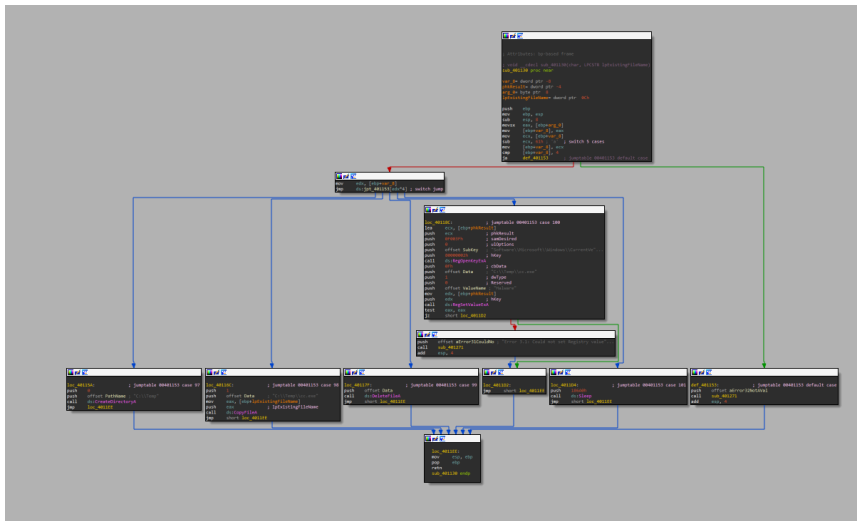


Figure 3.2: A screenshot from IDA Pro showing a control flow graph.

3.2 Function Calls

Functions (sometimes called *subroutines*) are an important part of modern programming languages. They allow splitting up code into reusable blocks. Functions can take inputs, called *arguments*, and can return an output value. Executing the code in a function is done by *calling* the function.

In C++ (and most other programming languages), function calls are expressed using the following format: `function_name(arg 0, arg 1, ..., arg n)`. Function calls consist of the following *function call elements*:

1. The function name.
2. The number of arguments.
3. The value and type of each argument.

For example, the function call `sum(1, 2)` to the function in [Listing 3.4](#) tells us that:

1. The function name is “sum”.
2. The function takes two arguments.
3. The first argument is an `int`, 1.
4. The second argument is also an `int`, 2.

```

1 int sum(int a, int b){
2     return a + b;
3 }

```

Listing 3.4: A C function that adds to integers together.

3.2.1 Calling Conventions

Functions are supported in assembly, through the `call` and `ret` instructions (discussed in [section 3.1.3](#)). However, assembly does not explicitly support passing arguments to functions. Instead, arguments have to be stored in memory or registers before a function call such that the function will be able to access them. A *calling convention* defines how arguments are passed to a function, on the assembly level. More specifically, a calling convention defines:

- How arguments are passed to the callee. This can be on the stack, in registers, or both.
- The order in which arguments are passed to the callee.
- Whether the callee or caller cleans up the stack after the callee is finished.
- How return values are passed to the caller.

Calling conventions are specific to a function, which means that multiple calling conventions can be used throughout a single executable.

As we will see in [subsection 3.2.2](#), calling conventions are important when analyzing function calls.

There are many calling conventions in x86. We will discuss the three most prominent: `cdecl`, `stdcall`, and `fastcall`.

Cdecl

The `cdecl`² convention is the default calling convention used by C. Arguments are pushed to the stack, right to left (i.e. the first argument is pushed to the stack last). The caller cleans up the stack. The return value is passed via `eax`.

The example C code and corresponding assembly³ in [Listing 3.5](#) and [Listing 3.6](#) shows `cdecl` in practice.

1. `main` pushes the arguments to the stack (lines 7 and 9).
2. `main` calls `sum` (line 10).
3. `sum` runs and saves its result in `eax` (line 4).
4. `main` cleans up the stack after the call (line 11), by lowering the top of the stack by 8 (the size of two integers).

²<https://docs.microsoft.com/en-us/cpp/cpp/cdecl?view=msvc-170>

³Compiled without any optimizations.

```

1 int sum(int a, int b) { push    ebp
2     return a + b;      mov     ebp, esp
3 }                      mov     eax, [ebp + 8]
4                        add     eax, [ebp + 12]
5                        pop     ebp
6                        retn

```

Listing 3.5: The C code and assembly of a function that uses `cdecl`.

```

1 int main() {           push    ebp
2     int x = 1;         mov     ebp, esp
3     int y = 2;         sub     esp, 8
4     return sum(x, y);  mov     [ebp - 8], 1
5 }                     mov     [ebp - 4], 2
6                       mov     eax, [ebp - 4]
7                       push    eax
8                       mov     ecx, [ebp - 8]
9                       push    ecx
10                      call    sum
11                      add     esp, 8
12                      mov     esp, ebp
13                      pop     ebp
14                      retn

```

Listing 3.6: The C code and assembly of a function calling the function from Listing 3.5 using `cdecl`.

Stdcall

The `stdcall`⁴ convention is used to call Win32 API functions. It is similar to `cdecl`, however, in `stdcall`, the callee cleans up the stack after it is done.

Fastcall

`fastcall`⁵ is a calling convention designed by Microsoft. It is meant to offer a performance boost over `cdecl` and `stdcall` by passing some arguments via registers. The first two arguments that fit in a register are passed in `ecx` and `edx`, respectively. All other arguments are pushed to the stack right to left. In `fastcall`, the callee is responsible for cleaning up the stack after it is done.

In Listing 3.7, and Listing 3.8 we see C code and assembly similar to the code in section 3.2.1. `sum` in Listing 3.7 uses `fastcall`⁶.

1. `main` stores the first two arguments to `edx` and `ecx` (lines 9 and 10).
2. `main` pushes the third argument to the stack (line 8).

⁴<https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=msvc-170>

⁵<https://docs.microsoft.com/en-us/cpp/cpp/fastcall?view=msvc-170>

⁶This can be explicitly specified in C code by adding the `fastcall` keyword to the function declaration. For example, `int __fastcall sum(int a, int b, int c)`.

3. `main` calls `sum` (line 11).
4. `sum` runs and saves its result in `eax` (line 8).
5. `sum` cleans up the stack (line 11) by lowering the top of the stack by 4 (the size of one integer). This implicitly happens in the `retn 4` instruction.

```

1  int sum(int a,int b,int c){ push    ebp
2      return a + b + c;      mov     ebp, esp
3  }                          sub     esp, 8
4                              mov     [ebp - 8], edx
5                              mov     [ebp - 4], ecx
6                              mov     eax, [ebp - 4]
7                              add     eax, [ebp + 8]
8                              add     eax, [ebp + 8]
9                              mov     esp, ebp
10                             pop     ebp
11                             retn    4

```

Listing 3.7: The C code and assembly of a function that uses fastcall.

```

1  int main() {               push    ebp
2      int x = 1;             mov     ebp, esp
3      int y = 2;             sub     esp, 12
4      int z = 3;             mov     [ebp - 12], 1
5      return sum(x, y, z);   mov     [ebp - 8], 2
6  }                          mov     [ebp - 4], 3
7                              mov     eax, [ebp - 4]
8                              push    eax
9                              mov     edx, [ebp - 8]
10                             mov     ecx, [ebp - 12]
11                             call    sum
12                             mov     esp, ebp
13                             pop     ebp
14                             retn

```

Listing 3.8: The C code and assembly of a function calling the function from [Listing 3.7](#) using fastcall.

3.2.2 Decompiling Function Calls

Compiling is the process of translating source code into executable machine code. *Decompiling* is the reverse process: translating machine code back into source code. Decompilers are important in reverse engineering software (especially malware analysis), because, if done well, they allow a far better understanding of the functionality of an executable binary. As information about the source code is lost during compilation, it is often not possible to perfectly translate a binary into its source code. Decompilation often relies on heuristics to structure machine code into source code.

In [chapter 6](#), we will use a decompiler (from IDA Pro) to decompile function calls. As function calls are not standardized in assembly, decompiling function calls is a complex process, which is also heavily dependent on heuristics. When we find a `call` instruction that we want to decompile, there are two steps to reconstructing the function call:

1. Determine the calling convention.

As we discussed in [subsection 3.2.1](#), there are various calling conventions that are commonly used (e.g. `cdecl` is the default calling convention in C++). However, compilers have total freedom in deciding how arguments are passed to a function in assembly and compilers for different programming languages often use custom calling conventions.

The two important characteristics that differentiate calling conventions from one another are how arguments are passed and how the memory is cleaned up after the function call has finished. If we can discover how these two characteristics are implemented for a function call, we determine what calling convention is used.

2. If we know the calling convention, we know how arguments are passed to the callee and in what order they are passed.

We can use this information to backtrack from the `call` instruction and detect instructions that place data in the locations (i.e. registers or the stack) that are used to pass the arguments.

For example, if we know that an argument is passed via the `ecx` register, we will look for instructions that store data in the `ecx` register.

However, as arguments can be computed at runtime and compilers optimize memory usage, it is often impossible to reconstruct an argument.

An Example of Decompiling a Function Call

Let's look at an example of decompiling a function call. [Listing 3.9](#) shows a simple example of a function call (to call the `sum` function in [Listing 3.4](#)).

```

1 int main() {           push    ebp
2     int x = 1;         mov     ebp, esp
3     int y = 2;         sub     esp, 8
4     return sum(x, y);  mov     [ebp - 8], 1
5 }                      mov     [ebp - 4], 2
6                      mov     eax, [ebp - 4]
7                      push    eax
8                      mov     ecx, [ebp - 8]
9                      push    ecx
10                     call    <address>
11                     add     esp, 8
12                     mov     esp, ebp
13                     pop     ebp
14                     ret     0

```

Listing 3.9: The C and assembly code of a function that calls [Listing 3.4](#) (using cdecl).

The first step is to determine the calling convention. We do this by answering how arguments are passed and whether the callee or the caller cleans up the stack.

How are arguments passed? In the instructions right before the `call` instruction, from lines 4 to 9, we see that two integers (1 on line 4 and 2 on line 5) are pushed to the stack. This is a good indication that either `cdecl` or `stdcall` are used.

Does the callee or the caller clean up the stack? Right after the function call, on line 11, we see that the stack pointer is reduced by 8 bytes (the size of two integers). This tells us that the `main` function (i.e. the caller) cleans up the stack. A common calling convention that requires the caller to clean up the stack after a function call is `cdecl`.

By answering these two questions, we have determined that the calling convention `cdecl` is most likely used.

The second step is to find the arguments that are passed to the function. As we noted earlier, we see that two integers are pushed to the stack (in the instructions on lines 4 to 9). Backtracking from the call signature, we first encounter 2 being pushed on the stack and after that we encounter 1 being pushed on the stack.

This tells us what the function call looks like `?(1, 2)`. The only part that we do not know is the function name. And because compilers often do not use function names from source code, we will not be able to reconstruct the function name.

If a callee is not part of the binary itself, but available in a library, the name might be available. We will discuss this in more detail in [section 3.3.1](#).

3.3 Windows

In this thesis, we focus on malware that is made for Windows. In this section, we go over three parts of the Windows operating system that are important to malware.

3.3.1 The Windows API

To interact with the Windows operating system (e.g. write to a file), applications need to call functions provided by Windows. These functions (and related data structures) are collectively called the Windows API. The API is documented online⁷.

In [chapter 7](#) and [chapter 5](#), we will see that malware calls specific API functions to achieve persistence.

[Listing 3.10](#) shows a simple example program that writes text to a file⁸. It calls three functions to interact with Windows: `CreateFile`⁹, `WriteFile`¹⁰ and `CloseHandle`¹¹.

⁷https://docs.microsoft.com/en-us/windows/win32/api/_winprog/

⁸The Windows API uses its own types to represent C types. For example, `DWORD` is the API type for a 32-bit unsigned integer.

⁹<https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew>

¹⁰<https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile>

¹¹<https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle>


```

1 #include <iostream>
2 #include <windows.h>
3
4 int main() {
5     HANDLE hFile;
6
7     LPCWSTR filename = L"test.txt";
8     LPCWSTR buffer = L"Hello World";
9     DWORD szBuffer = wcslen(buffer) * sizeof(WCHAR);
10
11     hFile = CreateFile(
12         filename,
13         GENERIC_WRITE,
14         0,
15         NULL,
16         CREATE_ALWAYS,
17         FILE_ATTRIBUTE_NORMAL,
18         0);
19     WriteFile(
20         hFile,
21         (LPVOID) buffer,
22         szBuffer,
23         0,
24         NULL);
25     CloseHandle(hFile);
26
27     return 0;
28 }

```

Listing 3.10: An example of creating and writing to a file using the Windows API.

Dynamic-link Libraries

The functions in the Windows API are provided in dynamic-link libraries (DLLs), executables that do not run by themselves. They export functions that can be used by other applications. Dynamic-link libraries offer multiple advantages over statically linked libraries. Most notably, functions do not have to be compiled into every executable that uses them and need to be loaded into memory only once.

Calling API Functions

As we can see in [Listing 3.10](#), Windows API functions are called like any other function. The process of matching a function call to the correct address is handled by Windows and transparent to developers.

Executables have an *Import Address Table* (IAT), which is used to translate external function calls in the executable to the actual addresses of the functions in memory. When a function in a DLL is called, the call in-

struction jumps to an address in the IAT, which in turn jumps to the actual address of the function. At load-time (i.e. when an executable is loaded into memory before executing it), Windows adds the actual memory addresses of the functions to the IAT. This process is called *load-time dynamic linking* [23].

It is also possible to load external functions at run-time (i.e. during the execution of an application), called *run-time dynamic linking*. This requires developers to write their own code that finds the right address to a function at run-time. The Windows API provides two functions that help developers do this: `LoadLibrary`¹² and `GetProcAddress`¹³. They work like this:

1. `LoadLibrary` is called to get a handle on the DLL that contains the required function.
2. `GetProcAddress` is called to search for the required function using the handle to the DLL.

Malware authors prefer to load functions at run-time because the functions are not added to the IAT, which hides the external functions they make use of.

A Comparison of Load-Time Dynamic Linking & Run-time Dynamic Linking

Appendix B contains the source code of two executables. The executables are semantically equivalent, but Listing B.1 uses direct function calls to Windows API and Listing B.2 uses run-time dynamic linking. Both executables perform the following steps:

1. Open the Windows Registry key `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion`, by calling `RegOpenKeyEx`¹⁴.
2. Get the data of the value `ProductName` in the Registry key, by calling `RegQueryValueEx`¹⁵.
3. Close the Registry key, by calling `RegCloseKey`¹⁶.

In Figure 3.3, we see the Assembly that is used to call `RegQueryValueEx` in both executables. We can see that the disassembler detected which function is being called in Figure 3.3a, but not in Figure 3.3b. The `call` instruction in Figure 3.3b does not jump to a static address, but to an address that is loaded into the `eax` register.

¹²<https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya>

¹³<https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getProcAddress>

¹⁴<https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regopenkeyexw>

¹⁵<https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regqueryvalueexw>

¹⁶<https://docs.microsoft.com/en-us/windows/win32/api/winreg/nf-winreg-regclosekey>

```

lea     eax, [esp+110h+phkResult]
mov     [esp+110h+cbData], 100h
push    eax                ; phkResult
push    20119h             ; samDesired
push    0                  ; ulOptions
push    offset SubKey      ; "SOFTWARE\\Microsoft\\Windows NT\\Curren"...
push    80000002h          ; hKey
call    ds:RegOpenKeyExW

```

(a) A (load-time dynamically linked) call to `RegOpenKeyExW` (disassembled by IDA Pro).

```

lea     ecx, [esp+120h+var_110]
push    ecx
push    20119h
push    0
push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows NT\\Curren"...
push    80000002h
call    eax

```

(b) A run-time dynamically linked call to `RegOpenKeyExW` (disassembled by IDA Pro).

Figure 3.3: A comparison of a load-time dynamically linked call and a run-time dynamically linked call to `RegOpenKeyExW`.

In Figure 3.4, we see part of the import table (decoded by IDA Pro) of the two executables. In Figure 3.4a, we see the three API functions, as expected. However, in Figure 3.4b we do not see the API functions, because the executable loads the functions itself.

Address	Ordinal	Name	Library
00403000		RegOpenKeyExW	ADVAPI32
00403004		RegCloseKey	ADVAPI32
00403008		RegQueryValueExW	ADVAPI32
00403010		GetModuleHandleW	KERNEL32
00403014		IsDebuggerPresent	KERNEL32
00403018		InitializeSListHead	KERNEL32
0040301C		GetSystemTimeAsFileTime	KERNEL32

(a) The imports of Listing B.1 using load-time dynamically linked calls.

Address	Ordinal	Name	Library
00403000		GetLastError	KERNEL32
00403004		LoadLibraryW	KERNEL32
00403008		GetProcAddress	KERNEL32
0040300C		FreeLibrary	KERNEL32
00403010		IsDebuggerPresent	KERNEL32
00403014		InitializeSListHead	KERNEL32
00403018		GetSystemTimeAsFileTime	KERNEL32

(b) The imports of Listing B.2 using run-time dynamically linked calls.

Figure 3.4: A comparison of the import address table of two executables.

3.3.2 The Windows Registry

The Windows Registry¹⁷ is a database of configuration settings for both the OS and applications. It is used to store almost all settings of the Windows operating system (e.g. the current Windows Version and the configuration of each service).

¹⁷ <https://docs.microsoft.com/en-us/windows/win32/sysinfo/registry>

Malware uses the Registry as a source of information (e.g. to find the current version of the OS) and to achieve persistence¹⁸ (as we see in [section 7.1](#)).

The Structure of the Windows Registry

The Registry has a tree-hierarchy that consists of *keys* and *values*. A key stores other keys (referred to as *subkeys*) and values. Values are a combination of a name, data, and the type of the data, similar to files. A path of subkeys is referenced using a backslash as a delimiter, similar to Windows directory paths.

Let's look at an example Registry key: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`. This path contains five keys: `HKEY_LOCAL_MACHINE`, `SOFTWARE`, `Microsoft`, `Windows NT` and `CurrentVersion`. `SOFTWARE` is a subkey of `HKEY_LOCAL_MACHINE`, `Microsoft` is a subkey of `SOFTWARE`, etc.

Registry paths always start with a root key. There are five root keys:

- `HKEY_LOCAL_MACHINE` (HKLM): Stores settings that apply to the whole operating system and all users.
- `HKEY_USERS` (HKU): Stores user-specific settings. The direct subkeys of HKU are the identifiers of each user.
- `HKEY_CURRENT_USER` (HKCU): A virtual key that points to the subkey of the current user in HKU.
- `HKEY_CURRENT_CONFIG` (HKCC): Stores information on the current hardware configuration.
- `HKEY_CLASSES_ROOT` (HKCR): Stores information on file extension associations.

The most important root keys for malware are HKLM and HKCU, as they contain the most useful information and control the most useful settings (e.g. which executables are automatically started at boot time).

CRUD Operations on the Windows Registry

There are multiple ways to interact with the Windows Registry (i.e. perform CRUD¹⁹ operations on the Windows Registry):

¹⁸Fileless malware, a type of malware that does not depend on executable files, often uses the Registry to store its executable code [17].

¹⁹CRUD stands for the four basic operations on databases and storage: Create, Read, Update and Delete.

- The Windows API: The Windows API exposes multiple functions that can be used to read or manipulate the Registry. For example, `RegOpenKeyEx` and `RegQueryValueEx` used in [section 3.3.1](#).
- Registry Editor: Windows provides a GUI application that users can use to view and change the Registry.
- `reg.exe` commands²⁰: Windows also provides a command-line application that can be used to view and change the Registry from the command line.
- `.reg` files: Windows support special script-like files that contain values for Registry keys. When these are executed, Windows automatically changes the Registry. [Listing 3.11](#) shows the contents of an example `.reg` file.

```

1      Windows Registry Editor Version 5.00
2
3      [HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
4      "startup value"="C:\Windows\System32\cmd.exe"
5

```

Listing 3.11: An example `.reg` file.

Malware most often uses the Windows API to edit the Registry directly [\[34\]](#).

As some registry values control global settings or settings that impact security, many keys require administrative privileges to change (e.g. all keys under HKLM).

3.3.3 Scheduled Tasks & Services

Windows supports running processes in the background without user interaction. This is useful for processes that should always be active, but do not require user interaction (e.g. antivirus software or hardware drivers). As we will see in [section 7.3](#) and [section 7.4](#), malware also uses background tasks for persistence.

A Windows service²¹ is a process that is started automatically when a Windows system is started (or other another Windows event happens) and runs in the background. Services can run even before a user has logged in, because they can run under the privileges of special system users.

Scheduled Tasks²² are similar to services. A scheduled task starts an application at a specific event (e.g. when a user logs in), at a pre-defined time, or on a repeating schedule, without any user interaction.

²⁰ <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/reg>

²¹ <https://docs.microsoft.com/en-us/windows/win32/services/services>

²² <https://docs.microsoft.com/en-us/windows/win32/taskschd/task-scheduler-start-page>

Services and Scheduled Tasks can run as any user account. However, as some services need to run even when no user is logged in, they can also run as one of three special accounts²³:

- **LocalService**²⁴: An account with minimal privileges.
- **NetworkService**²⁵: An account with minimal privileges with permission to authenticate to remote servers.
- **LocalSystem**²⁶: An account with full privileges on the local machine. This account is especially interesting to malware, as it grants full control of the machine.

Creating a service or scheduled task requires Administrator privileges, as they can be started using the **LocalSystem** user, which grants full control of the host.

Creating Services

There are multiple ways to create a service:

- Windows provides a GUI (called “Services”) that Administrator users can use to create, list, and modify services.
- The Windows API provides a function to create a Windows service: **CreateService**²⁷.
- Windows provides a command-line application (**sc.exe**²⁸) that allows users with Administrator privileges to create services from the command line.
- Like most configuration settings on Windows, the configuration of services is stored in the Registry. The configuration of services is stored in the key **HKLM\SYSTEM\CurrentControlSet\services**.

Creating Scheduled Tasks

There are multiple ways to create scheduled tasks in Windows:

- The Windows Task Scheduler is a GUI interface that Administrator users can use to create, list, and modify scheduled tasks.

²³ <https://docs.microsoft.com/en-us/windows/win32/services/service-user-accounts>

²⁴ <https://docs.microsoft.com/en-us/windows/win32/services/localservice-account>

²⁵ <https://docs.microsoft.com/en-us/windows/win32/services/networkservice-account>

²⁶ <https://docs.microsoft.com/en-us/windows/win32/services/localsystem-account>

²⁷ <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-createservicea>

²⁸ <https://docs.microsoft.com/en-us/windows/win32/services/configuring-a-service-using-sc>

- It is possible to create a scheduled task with the Windows API²⁹. To do this, one first needs to connect to the service used by the Scheduled Tasks. This is done using the `CoCreateInstance`³⁰ function. This function is called with `CLSID_TaskScheduler` as its first argument and `IID_ITaskService` as its fourth.
- `schtasks.exe`³¹ is a command-line application that allows users with Administrator privileges to create Scheduled Tasks from the command line.
- The predecessor of `schtasks.exe` is `at.exe`³². It works similarly to `schtasks.exe`, in that it can be used to create Scheduled Tasks from the command line. Although it is deprecated, it is still available.

²⁹ <https://docs.microsoft.com/en-us/windows/win32/taskschd/daily-trigger-example--c->

³⁰ <https://docs.microsoft.com/en-us/windows/win32/api/combaseapi/nf-combaseapi-cocreateinstance>

³¹ <https://docs.microsoft.com/en-us/windows/win32/taskschd/schtasks>

³² <https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/at>

Chapter 4

Background on Malware & Malware Analysis

In [chapter 3](#) we went over the background of the x86 architecture and Windows because malware is most prevalent on Windows. In this chapter, we look at how malware and malware analysis work in general. In [chapter 7](#), we discuss persistence, an important step in the malware life cycle.

Malware comes in many shapes and sizes, depending on its goals. However, most malware follows a similar trajectory to achieve its goals. Lockheed Martin developed a model of an intrusion of a computer system or network, the Cyber Kill Chain [\[15\]](#). This kill chain can also be used to model the life cycle of a malware infection:

1. **Reconnaissance:** The malicious actor gathers information on the target they want to compromise.
2. **Weaponization:** The malicious actor develops malware to infect the target.
3. **Delivery:** The malware is delivered to the target environment (e.g. by email or via a website that the target often visits).
4. **Exploitation:** The malware is executed inside the target environment (e.g. by using an exploit).
5. **Installation:** Once the malware is successfully executed on the victim's machine, it does an initial setup to maintain persistence.
6. **Command & Control (C2):** After the malware has established a solid foothold on the target machine, the malware beacons (i.e. signals) to a server owned by the malicious actor (i.e. the command & control server) that it has successfully infected a machine. This server may give additional commands the malware should perform.

7. **Actions on Objectives:** At this point in the infection, the malware has nested itself in the target environment and is ready for its main objective. This depends on the goals of the actor behind the malware. Some common tasks are:

- Privilege escalation: The malware will exploit a vulnerability or misconfiguration on a system to gain more access.
- Lateral movement: If the infected machine is part of a larger network, malware will try to infect other machines on the network.
- Data exfiltration: If the infected system contains valuable information, the malware will extract it and send it to the C2 server.
- Data encryption: In recent years, criminal hackers have started encrypting the files on infected machines and demanding payment for the decryption key [30].

This thesis focuses on how malware achieves persistence, the goal during the installation phase in the life cycle. In chapter 7, we discuss which techniques malware commonly implements to achieve persistence.

The MITRE ATT&CK framework¹ (based on the Cyber Kill Chain) is a taxonomy of offensive techniques and tactics used by hackers to intrude and infect computer systems. Persistence is one of the tactics it describes². We will be using it as a point of reference for the techniques we discuss in chapter 7.

4.1 Malware Analysis

Malware analysis is the research field that focuses on extracting information from malware (and the infrastructure supporting malware). As most malware is created as a Windows executable binary, malware analysis has a heavy focus on the analysis of Windows executables.

The goals of the analysis of a malware sample range from purely technical (e.g. what is the sample capable of doing?) to looking at an actor and their goals as a whole (e.g. why is an actor trying to infect specific targets?).

There are two types of malware analysis [34]:

- *Static* malware analysis: Useful information is extracted from a malware sample to get an idea of how it works. For example, a simple, useful technique is to extract strings from a malware binary as these give clues about its inner workings (e.g. URLs that the malware contacts). An important part of static analysis is translating the machine

¹<https://attack.mitre.org>

²<https://attack.mitre.org/tactics/TA0003/>

code in the binary to human-readable assembly code (i.e. disassembling) to see what a malware sample does on a low-level, without running it.

- *Dynamic* malware analysis: A sample of malware is executed in a sandbox (i.e. a controlled environment) and its actions are monitored. Modern malware often employs techniques to detect whether it is running in a sandbox to evade dynamic analysis.

In this thesis, we focus on static analysis.

4.1.1 The Advantages & Limitations of Static Analysis

Both static and dynamic malware analysis have their advantages and limitations [10]. As static analysis does not run the samples that are being analyzed, it is relatively safe and scalable, as you do not have to set up and clean a testing environment for each sample you test.

The biggest drawback of static malware analysis is that it requires disassembling and understanding the binary code inside an executable. This is a complex and hard problem to solve, as it is often only clear what a program does when it is run³. Malware authors obfuscate their malware to make it even harder to parse (e.g. by interleaving code and data throughout a binary).

4.1.2 IDA Pro

Disassemblers are programs that take a binary and parse the binary code into human-readable assembly code. They often also perform code analysis on the assembly. For example, to detect cross-references and Windows API imports.

The best-known reverse engineering toolkit is IDA Pro by Hex-Rays⁴ [7]. Other popular options include Ghidra⁵ and Radare2⁶.

IDA Pro is a reverse engineering toolkit with many advanced features:

- Standard static analysis tools such as a disassembler, detection of strings and detection of imports in the import table (discussed in [section 3.3.1](#)).
- Graph view: IDA Pro allows you to visualize the basic blocks in a function as a graph. For example, see [Figure 3.2](#).

³The Halting Problem shows us that creating a general algorithm to determine the functionality of a program without running the program is undecidable [36]. This, of course, also holds for malware [32].

⁴<https://www.hex-rays.com/ida-pro/>

⁵<https://ghidra-sre.org/>

⁶<https://rada.re/n/>

- Support for many types of binaries, operating systems, and instructions sets.
- Decompiler: One of the most important parts of IDA Pro is the state-of-the-art decompiler that can be used to decompile a binary into C-like pseudocode.
- Debugging: Support for multiple debuggers to closely observe what an application does, while it is running. As this runs the executable, it is used for dynamic analysis of binaries.

Plugins

IDA Pro provides an API that can be used to develop plugins⁷. Plugins are able to extend and use virtually every feature in IDA Pro, making them a powerful tool to automate part of malware analyses. Plugins are written in either IDC⁸ (a variant of C specifically made for IDA Pro) or Python⁹.

A drawback of developing IDA Pro plugins is the limited documentation of the API¹⁰. Because of this, a common practice in IDA Pro plugin development is looking for existing, open source plugins and see how they work, instead of reading the API documentation.

In [chapter 6](#), we implement the ideas of [chapter 5](#) in an IDA Pro plugin.

⁷ <https://www.hex-rays.com/products/ida/tech/plugin/>

⁸ <https://www.hex-rays.com/products/ida/support/idadoc/157.shtml>

⁹ <https://github.com/idapython/src>

¹⁰ https://www.hex-rays.com/products/ida/support/idapython_docs/

Chapter 5

Call Signatures

In this chapter, we present *Call Signatures*: patterns that allow us to search for the function name and arguments of function calls.

In [chapter 6](#), we implement an IDA Pro plugin that searches for function calls that match the patterns in Call Signatures. Taken together, the plugin and Call Signatures allow us to search for specific function calls in binaries. By not only constraining the function name but also constraining the arguments of function calls, we can find specific uses of a function (e.g. opening a specific Registry key)

In [chapter 8](#), we provide several Call Signatures that match function calls that are used to implement persistence techniques (which are discussed in [chapter 7](#)). In [chapter 9](#), we show that it is possible to detect malware capabilities using Call Signatures and our plugin, by detecting persistence techniques in real-world malware samples.

We define Call Signatures and discuss their syntax in detail in [section 5.1](#). We look at some examples of Call Signatures in [section 5.2](#)

Call Signatures define constraints on *function call elements*: the function name, the number of arguments, and arguments.

Call Signatures consists of multiple *rules*, expressions that constraint one function call element. For example, if we want to search for the function call `sum(0, 3)`, we could define the following rules:

1. The function name equals “sum”.
2. The function takes two arguments.
3. The first argument equals 0.
4. The second argument equals 3.

We use the following definitions in this section:

- **Function call element:** One of the following elements that are part of a function call: the function name, the number of arguments and the arguments. See [section 3.2](#) for more details.
- **Rule:** A constraint on a function call element. For example, the function name should contain “sum”.
- **Call Signature:** A combination of rules that together constraint a function call. Note the distinction that Call Signatures define constraints on *function calls* and rules define constraints on *function call elements*.

5.1 The Syntax of Call Signatures

We write Call Signatures in YAML¹, a popular format in software development because it is easily readable by both humans and code.

[Listing 5.1](#) shows an example of a Call Signature that can be used to search for function calls used in a Registry-based persistence technique. We will discuss the persistence technique in more detail in [section 7.1](#).

In essence, the example in [Listing 5.1](#) describes the following two possible function calls:

- `RegCreateKey(0x80000001, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", ?)`
- `RegOpenKey(0x80000001, "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run", ?)`

¹<https://yaml.org/>

```

1 ---
2
3 signature:
4   technique: "RPO"
5   description: >
6     This Call Signature can be used to
7     search for calls to RegCreateKey or RegOpenKey,
8     that are used to implement RPO.
9   rules:
10     - element: "function name"
11       contains_in:
12         - "RegCreateKey"
13         - "RegOpenKey"
14
15     - element: "number of arguments"
16       equals: 3
17
18     - element: "argument"
19       argument_index: 0
20       equals: 0x80000001
21
22     - element: "argument"
23       argument_index: 1
24
25     contains: "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run"

```

Listing 5.1: An example Call Signature. Colors are added for clarity.

This example in [Listing 5.1](#) shows the general structure of a Call Signature:

- The three dashes at the top signify the start of a YAML file.
- In blue, we see `signature`. All parts of the Call Signature fall under this YAML key.
- In green, we see two keys that give some descriptive information about the Call Signature. First, we see a `technique` key, that allows us to link this Call Signature to a specific technique (i.e. from [chapter 7](#)). Secondly, we see a `description` key that allows us to describe what the Call Signature is looking for.
- In yellow, we list of rules under the `rules` key. In [section 5.1.1](#), we will discuss what each detail of a rule means.

5.1.1 Rules

A rule defines a constraint on either the *value* or the *type* of a function call element. It consists of the following:

- The function call element that it constrains.
- Either one of:
 - The type that the function call element should be.
 - A value and an operator that should be compared to the function call element.

The Syntax of Rules

In [Listing 5.1](#), we saw how rules fit into the larger structure of a Call Signature. In this section, we will go into detail about each part of a rule. In [Listing 5.2](#), and [Listing 5.3](#) we see two (color-coded) examples of rules. [Listing 5.2](#) is a rule that defines a constraint on a value and [Listing 5.3](#) is a rule that defines a constraint on a type.

```
1 - element: "function name"
2   contains: "RegOpenKey"
```

Listing 5.2: An example rule defining a constraint on a value.

```
1 - element: "number of arguments"
2   type: "number"
```

Listing 5.3: An example rule defining a constraint on a type.

- In both rules, we first see a key value pair with an `element` key, in green. This key defines which function call element the rule defines a constraint on.
- In yellow, we see the operator (i.e. `contains`) and, in orange, we see the value that the operator should compare with the value of the function call element. This defines that “RegOpenKey” should be a substring of the function name (i.e. the function call element that the rule is about)
- In blue, we see a key-value pair that defines the type (in this case number) that the function call element (in this case the number of arguments) should have.

Function Call Elements

Function calls consist of multiple elements (i.e. the function name, the number of arguments, and each argument). To be able to define constraints on all these elements, the `element` key in a rule can have the following values:

- **function name:** The function name. [Listing 5.2](#) shows an example of such a rule.
- **number of arguments:** The number of arguments that are passed to the function. [Listing 5.3](#) shows an example of a rule constraining the number of arguments.
- **argument:** A specific argument, passed to the function=. This requires an additional `argument_index` YAML key to specify which argument the rule applies to. The first argument has the index 0 and the arguments are ordered from left to right. For example, [Listing 5.4](#) shows a rule that specifies that the second argument should be a string.

```
1 - element: "argument"
2   argument_index: 1
3   type: "string"
```

Listing 5.4: An example rule defining a constraints on the second argument.

- **any argument:** With this function call element, the rule does not apply to a specific argument, but all arguments. In other words, the constraint should be met by at least one argument. The rule in [Listing 5.2](#) says that at least one of the arguments passed during a function call should contain the string “example string”.

```
1 - element: "any argument"
2   contains: "example string"
```

Listing 5.5: An example rule defining a constraints on all arguments.

Operators

Constraints on the value of function call elements are defined using operators. An operator defines how the values (the value in the rule and the value of the function call element) are compared. These rules allow us to express constraints such as the function name should equal “RegCreateKey”.

The following operators are supported:

- **equals:** The two values match if they are exactly the same. [Listing 5.6](#) gives an example of a rule using the `equals` operator.

```
1 - element: "function name"
2   equals: "sum"
```

Listing 5.6: An example rule using the `equals` operator.

- **contains:** The values match if the value in the rule is a substring of the function call element. In other words, `contains: "example string"` matches if the function call element contains the string “example string”. [Listing 5.2](#) shows such a rule.

This comparison is case-insensitive.

- **in**: Matches, if the value of function call element is an element in a given list. For example, [Listing 5.7](#).

```

1 - element: "argument"
2   argument_index: 0
3   contains_in:
4     - 0x80000001
5     - 0x80000002

```

Listing 5.7: An example rule using the **in** operator.

- **contains_in**: Matches, if a string in a given list is a substring of the value of function call element. In other words, **contains**: `["string A", "string B"]` matches if the function call element contains either “string A” or “string B”.

For example, [Listing 5.7](#) shows a rule that specifies that at least one argument should contain “HKCU” or “HKEY_CURRENT_USER”.

```

1 - element: "any argument"
2   contains_in:
3     - "HKCU"
4     - "HKEY_CURRENT_USER"

```

Listing 5.8: An example rule using the **contains_in** operator.

Type

Constraints on the type of a function call element are defined by the **type** key. For example, these rules allow us to express constraints as the second argument should be a string.

[Listing 5.3](#) shows an example of such a rule. There are multiple available types:

- String
- Number
- Bytes: Sometimes it is useful to match the exact bytes of an argument (and not the string representation of the bytes). For example, in [section 8.2](#) and [section 8.4](#) we will see that this is useful to match structs with a constant value.

As YAML does not support bytes natively, we represent bytes as a hexadecimal string. [Listing 5.9](#) show such a rule.

```

1 - element: "argument"
2   argument_index: 1
3   equals: "48 65 6C 6C 6F 20 57 6F 72 6C 64"
4   type: "bytes"

```

Listing 5.9: An example rule specifying a bytes value.

Some types are pre-defined. For example, the number of functions is always a number and the function name is always a string.

In some cases, we want to specify a constraint on a value and type at the same time. For example, in [Listing 5.9](#), we give a string in the `equals` operator, but we explicitly specify that it should be interpreted as bytes.

5.2 Examples of Call Signatures

Let's look at an example of how to write Call Signatures. In [Listing 5.10](#), we see two simple C functions, `sum` and `add10`.

```
1 int sum(int a, int b){
2     return a + b;
3 }
4
5 int add10(int a){
6     return sum(10, b);
7 }
```

Listing 5.10: Two C functions.

Given these functions, we have two techniques to add the number 10 to another integer:

- **Add10**: Calling the `add10` function with an integer as its argument.
- **AddSum**: Calling the `sum` function with 10 as one of its arguments.

Let's say we want to search for these two techniques in a binary. We can use Call Signatures for this.

In [section 6.2](#) we will expand upon this example, by looking at how to search for function calls using these Call Signatures.

Writing a Call Signature for Add10 We know that calls to `add10` have the following specific properties that we can write rules for:

- The function name is “`add10`”. We need a rule that constrains the function name in the function call to “`add10`”:

```
1 - element: "function name"
2   equals: "add10"
```

- The function call has one argument, which we can constrain using a rule for the number of arguments:

```
1 - element: "number of arguments"
2   equals: 1
```

- The argument is an integer. Which we can capture in the following rule:

```

1 - element: "argument"
2   argument_index: 0
3   type: "number"

```

Combining all the above rules gives us the Call Signature in [Listing 5.11](#):

```

1 ---
2
3 signature:
4   technique: "Add10"
5   description: >
6     This Call Signature can be used to
7     search for calls to add10.
8   rules:
9     - element: "function name"
10       equals: "add10"
11
12     - element: "number of arguments"
13       equals: 1
14
15     - element: "argument"
16       argument_index: 0
17       type: "number"

```

Listing 5.11: A Call Signature for add10.

Writing a Call Signature for AddSum Function calls to `sum` that implement AddSum (i.e. that add 10 to another integer) are similar to those of `add10`, but in `sum` either argument can be 10. We use the following rules to constraint each property of such calls:

- The function name is “`sum`”. We can use the following rule for this:

```

1 - element: "function name"
2   equals: "sum"

```

- The function call has two arguments, which we can constrain using a rule for the number of arguments:

```

1 - element: "number of arguments"
2   equals: 2

```

- The first argument is an integer:

```

1 - element: "argument"
2   argument_index: 0
3   type: "number"

```

- The second argument is an integer:

```

1 - element: "argument"
2   argument_index: 1
3   type: "number"

```

- One of the arguments is 10. We can define a rule over all arguments (i.e. that succeeds if at least one argument matches), by using the **any** argument function call element:

```
1 - element: "any argument"
2   equals: 10
```

Combining all the above rules gives us the Call Signature in [Listing 5.12](#):

```
1 ---
2
3 signature:
4   technique: "AddSum"
5   description: >
6     This Call Signature can be used to
7     search for calls to sum.
8   rules:
9     - element: "function name"
10       equals: "sum"
11
12     - element: "number of arguments"
13       equals: 2
14
15     - element: "argument"
16       argument_index: 0
17       type: "number"
18
19     - element: "argument"
20       argument_index: 1
21       type: "number"
22
23     - element: "any argument"
24       equals: 10
```

Listing 5.12: A Call Signature for `sum`.

Chapter 6

Call Signatures Plugin

In [chapter 5](#), we introduced Call Signatures, patterns that describe function calls. In this chapter, we implement an IDA Pro plugin (discussed in [section 4.1.2](#)): *Call Signatures Plugin* (CSP for short). The goal of CSP is to search for function calls that match Call Signatures in a binary. In other words, Call Signatures and CSP are complementary: Call Signatures describe function calls and CSP searches for these function calls in a binary.

In [chapter 9](#), we experiment with this plugin by analyzing real-world world samples using Call Signatures from [chapter 8](#).

IDA Pro provides two advantages. First, IDA Pro contains a large suite of state-of-the-art static analysis tools, such as a disassembler and decompiler. IDA Pro plugins can build on these to automate part of malware analysis. Secondly, it allows malware analysts to easily integrate the plugin into their analysis routine, as many malware analysts will be using IDA Pro [\[7\]](#).

IDA Pro allows users to search for basic patterns (e.g. strings and bytes), but not for more complex structures (e.g. function calls). To search for specific structures or patterns, users need to implement their own plugin.

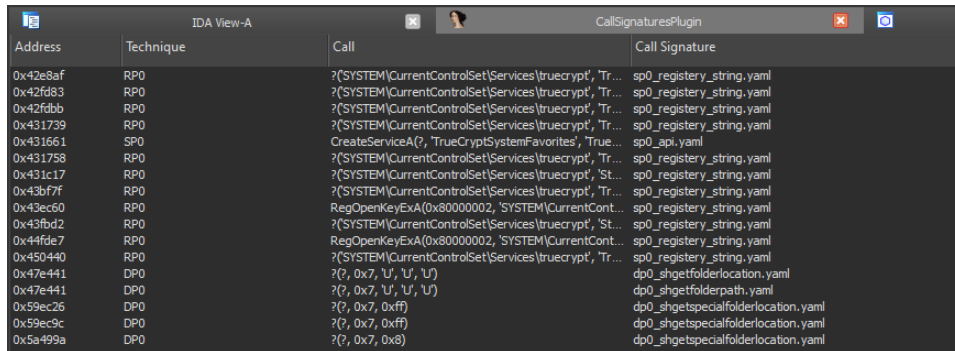
The source code is available at <https://github.com/joren485/CallSignaturesPlugin>.

The analysis of a binary using CSP is done in the following steps:

1. Before the plugin runs, IDA Pro disassembles the binary and extracts useful information. One of the most important steps the disassembler takes is to deduce the boundaries of each function in the binary.
2. CSP reads all Call Signatures (i.e. YAML files) in the plugin directory.
3. CSP loops over each function that IDA Pro has detected. For each function it takes the following steps:

- (a) If IDA Pro detects that the function matches a F.L.I.R.T. signature (see [chapter 2](#)), the function is skipped and CSP continues with the next function in the binary.
- Functions that match F.L.I.R.T. signatures are known uninteresting functions. They are either library functions (e.g. functions from the C standard library) or external functions (e.g. functions from a DLL). We skip these because we are interested in the functionality of the binary, not in the functionality of functions from libraries used by the binary.
- (b) CSP instructs IDA Pro to decompile the function. This transforms the assembly instructions of the function into C-like pseudocode.
- Decompilation relies heavily on probabilistic heuristics because a lot of information is lost during compilation. Because of this, decompilation might fail, because the decompiler is unable to reconstruct assembly into pseudocode.
- This decompilation includes reconstructing function calls as we discussed in [subsection 3.2.2](#).
- (c) CSP compares each function call to each Call Signature. This process is discussed in more detail in [section 6.1](#).
- If a function call matches a Call Signature, CSP writes to the IDA Pro log that a match was found.

[Figure 6.1](#) shows a screenshot of IDA Pro after CSP analyzed a sample¹. We can see an open tab that shows a list of detected function calls. For each function call, the plugin lists the address of the call instruction, the persistence technique, the decompiled call and the Call Signature that matched.



Address	Technique	Call	Call Signature
0x42e8af	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x42f883	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x42f8bb	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x431739	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x431661	SP0	CreateServiceA(?', TrueCryptSystemFavorites', 'True...	sp0_api.yaml
0x431758	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x431c17	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'St...	sp0_registry_string.yaml
0x43bf7f	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x43ec60	RP0	RegOpenKeyExA(0x80000002, '\SYSTEM\CurrentCont...	sp0_registry_string.yaml
0x43fbd2	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'St...	sp0_registry_string.yaml
0x44fde7	RP0	RegOpenKeyExA(0x80000002, '\SYSTEM\CurrentCont...	sp0_registry_string.yaml
0x450440	RP0	?(\SYSTEM\CurrentControlSet\Services\truecrypt', 'Tr...	sp0_registry_string.yaml
0x47e441	DP0	?(\?, 0x7, 'U', 'U', 'U)	dp0_shgetfolderlocation.yaml
0x47e441	DP0	?(\?, 0x7, 'U', 'U', 'U)	dp0_shgetfolderpath.yaml
0x59ec26	DP0	?(\?, 0x7, 0xff)	dp0_shgetspecialfolderlocation.yaml
0x59ec9c	DP0	?(\?, 0x7, 0xff)	dp0_shgetspecialfolderlocation.yaml
0x5a499a	DP0	?(\?, 0x7, 0x8)	dp0_shgetspecialfolderlocation.yaml

Figure 6.1: A screenshot of CSP.

¹4c01ffcc90e6271374b34b252fefb5d6fffd29f6ad645a879a159f78e095979

6.1 Call Signature Matching

For a function call to match a Call Signature, each of the rules in the Call Signature should succeed.

As discussed in [subsection 5.1.1](#), a rule is a constraint on a function call element (e.g. the function name). A rule can either succeed (i.e. the values match) or fail (i.e. the values do not match).

CSP iterates through each rule in a Call Signature and compares the constraint in the rule with the value of the relevant function call element.

The values of function call elements are often unknown, as information is lost during compilation (e.g. function names are not available in binaries) or determined at runtime (e.g. arguments passed by reference). Whether the value of the function call element is known, determines how the rule is applied to the function call element. The matching algorithm covers multiple situations:

- If the rule specifies a constraint on a value and the function call element value is known, the values are compared using the operator specified in the rule.
- If the rule specifies a constraint on a type and the function call element value is known, the rule succeeds if the type of the function call element value equals the type specified in the rule.
- If the function call element value is not known, the result depends on the function call element:

- If the function call element is the function name, the rule succeeds.

Because the disassembler can often not determine which function is being called (e.g. because the call is indirect), the function name is frequently unknown. To prevent many false negatives with rules that define a constraint on the function name, the rule succeeds.

- If the function call element is a specific argument or any argument, the rule fails.

The decompiler will not be able to determine the value of the majority of arguments passed in function calls (most often because the value is computed at runtime). To prevent too many false positives, the rule fails if the argument value is unknown.

6.2 Examples of Call Signatures Matching

In this section, we will look at an example of matching a function calls and Call Signatures. This is an example of the process laid out in [section 6.1](#).

In this example, we will use an example function ([Listing 6.1](#)) and corresponding Call Signature [Listing 6.2](#) from [section 5.2](#)

```
1 int sum(int a, int b){
2     return a + b;
3 }
```

Listing 6.1: A C function that adds two integers.

```
1 ---
2
3 signature:
4     technique: "AddSum"
5     description: >
6         This Call Signature can be used to
7         search for calls to sum.
8     rules:
9         - element: "function name"
10           equals: "sum"
11
12         - element: "number of arguments"
13           equals: 2
14
15         - element: "argument"
16           argument_index: 0
17           type: "number"
18
19         - element: "argument"
20           argument_index: 1
21           type: "number"
22
23         - element: "any argument"
24           equals: 10
```

Listing 6.2: A Call Signature for `sum`.

Let's say that IDA Pro identified the following function calls in a binary. To see if any of them match the Call Signature in [Listing 6.2](#), we need to iterate through all the rules and see if they succeed when applied to the function call. In the following examples, a “?” signifies a function call element with an unknown value.

- `sum(2, 3)`

1. The first rule succeeds because the function name equals “`sum`”.
2. The second rule succeeds because the call takes two arguments.
3. The third rule succeeds because the first argument is an integer (i.e. a number).
4. The fourth rule succeeds because the second argument is also an integer.
5. The fifth rule fails, because no argument is equal to 10.

As the fifth rule failed, this call is not a match.

- `?(5, 10)`

1. The first rule succeeds, as the function name is allowed to be unknown. This is explained in detail in [Listing 6.2](#).
2. The second rule succeeds because the call takes two arguments.
3. The third rule succeeds because the first argument is an integer.
4. The fourth rule succeeds because the second argument is also an integer.
5. The fifth rule succeeds because an argument (i.e. the second argument) is equal to 10.

As all rules succeeded, this call is a match for the Call Signature in [Listing 6.2](#).

- `sum(10, ?)`

1. The first rule succeeds because the function name equals “`sum`”.
2. The second rule succeeds because the call takes two arguments.
3. The third rule succeeds because the first argument is an integer.
4. The fourth rule failed, because the second argument is unknown. Unlike, the function name, rules will always fail on unknown arguments. This is explained in detail in [Listing 6.2](#).

As the fourth rule failed, this call is not considered a match and the fifth rule is not applied.

Chapter 7

Persistence Techniques

In this chapter, we will look more closely at *persistence*: the ways in which malware maintains access to infected systems. To test Call Signatures and CSP, we will use them to detect persistence in real-world malware samples (in [chapter 9](#)). Before we can do this, we need to better understand persistence and how it is implemented by malware. We choose to focus on persistence as it is common for malware to implement.

In [chapter 8](#), we will write Call Signatures that capture the function calls used to implement the techniques discussed in this chapter.

Persistence is achieved by using or altering system settings to automatically launch an executable whenever a specific startup event happens (e.g. the system booting or a user logging in). Windows provides multiple ways to automatically run applications on startup. This is a feature of Windows with legitimate use cases (e.g. starting Spotify or antivirus software automatically). These are generally known as *Autostart Extensibility Points* (ASEPs) [38].

There are hundreds of ASEPs [3], making it difficult to list all executables that are automatically run. Windows provides a tool, called Autoruns¹, that analyzes many ASEPs on a running Windows system and lists the executables and DLLs that are automatically started, but even this tool is not exhaustive [3].

However, not every ASEP is useful to malware, as many ASEPs depend on specific applications being installed or specific Windows features being enabled. Malware authors prefer to use ASEPs that are widely applicable (e.g. across all Windows systems) and reliable. Malware often uses multiple ASEPs, to increase their chances of maintaining access.

In this chapter, we discuss the ASEPs that are commonly² used by malware, which we refer to as persistence techniques. We subdivided these into

¹<https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns>

²According to well-known malware analysis sources such as Practical Malware Analysis [34].

four categories³: registry-based (section 7.1), directory-based (section 7.2), service-based (section 7.3) and scheduled task-based (section 7.4). We give each technique a unique identifier, to easily reference it later.

Besides a description of how a technique works, we also note under which user the executable is run and whether Administrator privileges are required to implement the technique.

As a point of reference, we provide the relevant MITRE ATT&CK ID (discussed in chapter 4) of each persistence technique. Note that multiple persistence techniques have the same ATT&CK ID, as they are categorized as the same ATT&CK technique. This shows that our classification is more fine-grained than the classification of the ATT&CK framework.

7.1 Registry-based Persistence Techniques

As discussed in subsection 3.3.2, the Registry is a database of configuration settings for Windows. Malware commonly use the following techniques to achieve persistence via the Registry.

RP0: Executables that are specified in a value in the Registry key `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` (or a subkey of this key) will run each time the current user logs in [9] [24].

Runs as: The user that logs in.

Administrator rights required: No

ATT&CK ID: T1547.001

RP1: The Registry key `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` works similarly to technique RP0, but the values are deleted when they are executed [9] [24].

Runs as: The user that logs in.

Administrator rights required: No

ATT&CK ID: T1547.001

RP2: `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` is the system-wide variant of the key in technique RP0. When a user logs in, all executables defined in values in this key (or a subkey of this key) are run [9] [34] [24].

Runs as: The user that logs in.

³There are more categories of persistence techniques, but these categories cover the commonly used techniques. See <https://attack.mitre.org/tactics/TA0003/> for a more complete taxonomy.

Administrator rights required: Yes

ATT&CK ID: T1547.001

RP3: The Registry key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce works similarly to [technique RP2](#), but the values are deleted when they are executed [9] [24].

Runs as: The user that logs in.

Administrator rights required: Yes

ATT&CK ID: T1547.001

RP4: The DLLs specified in the AppInit_DLLs value of HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows are loaded into every process that also loads User32.dll (most applications load User32.dll) [9] [34].

Runs as: The process that the DLL is loaded into.

Administrator rights required: Yes

ATT&CK ID: T1546.010

RP5: Windows runs `userinit.exe` when a user logs in to initialize the user session. This executable is specified in the `Userinit` value in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. The `Userinit` value can be changed to a comma-separated list of executables that will all be run when a user logs in [9] [6].

Runs as: The user that logs in.

Administrator rights required: Yes

ATT&CK ID: T1547.004

RP6: Windows runs `explorer.exe` when a user logs in to set up the graphical user interface. This executable is specified in the `Shell` value in HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon. Like [technique RP5](#), this value can be changed into a comma-separated list of executables [9] [6].

Runs as: The current user.

Administrator rights required: Yes

ATT&CK ID: T1547.004

7.2 Directory-based Persistence Techniques

Windows provides special startup directories. Executables (or shortcuts to executables) in these directories are run when a user logs in. Malware can simply place its executable file in such a directory, and it will be started every time a user logs in.

DP0: Windows provides a startup directory for every user [19].

The default path of this directory is `C:\Users\<user>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup`.

Runs as: The user that logs in.

Administrator rights required: No

ATT&CK ID: T1547.001

DP1: Windows also provides a system-wide startup directory. Files in these directories are run when any user logs in [19].

The default path of this directory is `C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup`.

Runs as: The user that logs in.

Administrator rights required: Yes

ATT&CK ID: T1547.001

7.3 Service-based Persistence Techniques

As discussed in [subsection 3.3.3](#), a Windows service is a background process that is started at boot time. Malware uses this to ensure that it is started as a background task.

SP0: Malware can create a service that starts its executable in the background on boot. As this service can run as the `LocalSystem` account, this would immediately give the malware full administrative privileges.

Runs as: Services can run as any user account, `LocalService`, `NetworkService`, or `LocalSystem`. See [subsection 3.3.3](#) for more detailed information on these accounts.

Administrator rights required: Yes

ATT&CK ID: T1043.003

As services are configured in the Windows Registry, there is some overlap between [technique SP0](#) and [section 7.1](#). However, as the goal is to start a service, we felt it best to add these into their own category.

7.4 Scheduled Task-based Persistence Techniques

Scheduled Tasks are a way to run applications at pre-defined events or times in Windows (discussed in [subsection 3.3.3](#)). Like services, malware can use them for persistence.

TP0: Malware can create a scheduled task that runs on boot or a schedule (e.g. every hour) to ensure persistence. As this task can run as the `LocalSystem` account, this would immediately give the malware full administrative privileges.

Runs as: Scheduled Tasks can run as a specific user, `LocalService`, `NetworkService`, or `LocalSystem`. See [subsection 3.3.3](#) for more detailed information on these accounts.

Administrator rights required: Yes

ATT&CK ID: T1053.005

Chapter 8

Call Signatures for Persistence Techniques

In this chapter, we will write Call Signatures (introduced in [chapter 5](#)) to describe function calls used in four of the commonly implemented persistence techniques (from [chapter 7](#)). In [chapter 9](#), we will use these Call Signatures to detect persistence techniques in real-world malware samples and to analyze the effectiveness of Call Signatures.

We introduced four categories of persistence techniques in [chapter 7](#) (directory-based, Registry-based, service-based, and scheduled task-based techniques). We first analyze what function calls are made to implement these persistence techniques, and we then look at how we can express these function calls as Call Signatures.

- In [section 8.1](#), we write Call Signatures for [technique RP0](#).
- In [section 8.2](#), we write Call Signatures for [technique DP0](#).
- In [section 8.3](#), we write Call Signatures for [technique SP0](#).
- In [section 8.4](#), we write Call Signatures for [technique TP0](#).

Finally, in [section 8.5](#), we reflect on writing Call Signatures.

8.1 Registry-based Persistence Techniques

In this section, we will discuss Call Signatures for [technique RP0](#), a technique that writes to (a subkey of) the Registry key `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`.

As we saw in [subsection 3.3.2](#), there are two common ways for applications to modify the Registry, namely the Windows API and the `reg.exe` command-line application.

8.1.1 Modifying the Registry via the API

To set the value in a Registry key using the Windows API, an application first needs to (create or) open it using one of the following functions (as seen in the examples in [Appendix B](#)):

- `RegCreateKey`
- `RegOpenKey`
- `RegCreateKeyEx`
- `RegOpenKeyEx`

All of these functions work similarly. The first argument takes a constant that specifies the root key. For example, the constant `0x80000001` represents `HKCU`. The second argument specifies the path of the key.

To detect [technique RP0](#) we need to find a call to a function where the first parameter is `0x80000001` and the second argument contains `"SOFTWARE\Microsoft\Windows\CurrentVersion\Run"`. [Listing 8.1](#) is a Call Signature for such calls.

```
1 ---
2
3 signature:
4   technique: "RP0"
5   description: >
6     This Call Signature can be used to
7     search for calls to RegCreateKey or RegOpenKey,
8     that are used to implement RP0.
9   rules:
10     - element: "function name"
11       contains_in:
12         - "RegCreateKey"
13         - "RegOpenKey"
14
15     - element: "number of arguments"
16       equals: 3
17
18     - element: "argument"
19       argument_index: 0
20       equals: 0x80000001
21
22     - element: "argument"
23       argument_index: 1
24       contains: "SOFTWARE\\Microsoft\\Windows\\CurrentVersion
25               \\Run"
```

Listing 8.1: A Call Signature for [technique RP0](#).

8.1.2 Modifying the Registry via Reg.exe

To modify the Windows Registry using `reg.exe`, one needs to run `reg.exe` (or just `reg`) command with `add` as the first argument. For example, the Windows command in [Listing 8.2](#).


```

1  reg add "HKEY_CURRENT_USER\Software\Microsoft\Windows\
   CurrentVersion\Run" /v WindowsUpdate /t REG_SZ /d "C:\Users
   \John\windowsupdate.exe"

```

Listing 8.2: A Windows Command that implements [technique RP0](#).

To match all situations in which such a command is used to implement [technique RP0](#), we are looking for the strings with the following substrings:

- “reg”
- “add”
- “HKCU” or “HKEY_CURRENT_USER”
- “SOFTWARE\Microsoft\Windows\CurrentVersion\Run”

As these strings will not always be part of the same argument, we need a Call Signature that has rules for each of them, separately. [Listing 8.3](#) does this.

```

1  ---
2
3  signature:
4    technique: "RP0"
5    description: >
6      This Call Signature can be used to search for
7      usage of reg.exe commands, that implement RP0.
8    rules:
9      - element: "any argument"
10        contains: "reg"
11
12      - element: "any argument"
13        contains: "add"
14
15      - element: "any argument"
16        contains_in:
17          - "HKCU"
18          - "HKEY_CURRENT_USER"
19
20      - element: "any argument"
21        contains: "SOFTWARE\\Microsoft\\Windows\\CurrentVersion
        \\Run"

```

Listing 8.3: A Call Signature for [technique RP0](#).

8.2 Directory-based Persistence Techniques

In this section, we will discuss Call Signatures for [technique DP0](#)

In [technique DP0](#), malware places an executable file (or a link to an executable file) in <User Directory>\AppData\Roaming\Microsoft\Win

dows\Start Menu\Programs\Startup to start the executable when the user logs in.

The path of special directories (e.g. the startup directories or a directory for temporary storage) might change between Windows versions. To solve this, the Windows API provides multiple functions that allow applications to resolve the path of special directories. This improves the backward compatibility of applications that use these special directories

To write to the startup directory, malware can either use the path directly or can use a Windows API function to resolve the path of the startup directory.

8.2.1 Using the Path of the Startup Directory Directly

If malware uses the path directly, we are looking for function calls that have the path as a string argument. The Call Signature in Listing 8.4 describes such function calls.

```
1 ---
2
3 signature:
4     technique: "DP0"
5     description: >
6         This Call Signature can be used to
7         search for calls that directly use the path of
8         the startup directory in a user's directory.
9     rules:
10         - element: "any argument"
11             contains: "AppData\\Roaming\\Microsoft\\Windows\\
                Start Menu\\Programs\\Startup"
```

Listing 8.4: A Call Signature for technique DP0.

8.2.2 Resolving the Path of the Startup Directory

From Windows Vista onwards, the API function to resolve the paths of special directories is `SHGetKnownFolderPath`¹.

This function takes a constant value as its first argument. This constant is a GUID², a 16 bytes unique identifier. For example, the identifier that is used to search for the startup directory of the current user is B97D20BB-F46A-4C97-BA10-5E3608430854³.

In a binary, a GUID is represented directly by the 16 bytes. Figure 8.1 shows how the disassembler of IDA Pro interprets the bytes in a GUID.

To detect calls to `SHGetKnownFolderPath` that resolve the path used in technique DP0, we need to write a rule that constrains the specific bytes of the GUID in the first argument, like in Listing 8.5.

¹https://docs.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetknownfolderpath

²<https://docs.microsoft.com/en-us/dotnet/api/system.guid>

³<https://docs.microsoft.com/en-us/windows/win32/shell/knownfolderid>

```

; const KNOWNFOLDERID rfid
rfid      dd 0897D2088h          ; Data1
                                   ; DATA XREF: _main+1Ffo
        dw 0F46Ah              ; Data2
        dw 4C97h               ; Data3
        db 08Ah, 10h, 5Eh, 36h, 8, 43h, 8, 54h; Data4

```

Figure 8.1: A GUID in a binary disassembled by IDA Pro.

```

1  ---
2
3  signature:
4      technique: "DPO"
5      description: >
6          This Call Signature can be used to
7          search for calls to SHGetKnownFolderPath.
8      rules:
9          - element: "function name"
10             equals: "SHGetKnownFolderPath"
11
12          - element: "number of arguments"
13             equals: 4
14
15          - element: "argument"
16             argument_index: 0
17             equals: "BB 20 7D B9 6A F4 97 4C BA 10 5E 36 08 43 08
18             54"
19             type: "bytes"

```

Listing 8.5: A Call Signature that matches a call to `SHGetKnownFolderPath`.

Deprecated API Functions

Before Windows Vista, there were multiple (now deprecated, but still available) functions available to resolve the paths of directories.

All these functions take a constant that represents the startup directory as an argument. This constant is called `CSIDL_STARTUP`⁴ and its value is 7.

⁴<https://docs.microsoft.com/en-us/windows/win32/shell/csidl>

We write Call Signatures for all of these functions:

- SHGetFolderPath⁵

```
1 ---
2
3 signature:
4   technique: "DP0"
5   description: >
6     This Call Signature can be used to
7     search for calls to SHGetFolderPath.
8   rules:
9     - element: "function name"
10       contains: "SHGetFolderPath"
11
12     - element: "number of arguments"
13       equals: 5
14
15     - element: "argument"
16       argument_index: 1
17       equals: 0x07
```

Listing 8.6: A Call Signature that matches a call to SHGetFolderPath.

- SHGetFolderLocation⁶

```
1 ---
2
3 signature:
4   technique: "DP0"
5   description: >
6     This Call Signature can be used to
7     search for calls to SHGetFolderLocation.
8   rules:
9     - element: "function name"
10       contains: "SHGetFolderLocation"
11
12     - element: "number of arguments"
13       equals: 5
14
15     - element: "argument"
16       argument_index: 1
17       equals: 0x07
```

Listing 8.7: A Call Signature that matches a call to SHGetFolderLocation.

⁵ https://docs.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetfolderpathw

⁶ https://docs.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetfolderlocation

- SHGetSpecialFolderPath⁷:

```

1 ---
2
3 signature:
4   technique: "DP0"
5   description: >
6     This Call Signature can be used to
7     search for calls to SHGetSpecialFolderPath.
8   rules:
9     - element: "function name"
10       contains: "SHGetSpecialFolderPath"
11
12     - element: "number of arguments"
13       equals: 4
14
15     - element: "argument"
16       argument_index: 2
17       equals: 0x07

```

Listing 8.8: A Call Signature that matches a call to SHGetSpecialFolderPath.

- SHGetSpecialFolderLocation⁸:

```

1 ---
2
3 signature:
4   technique: "DP0"
5   description: >
6     This Call Signature can be used to
7     search for calls to SHGetSpecialFolderLocation.
8   rules:
9     - element: "function name"
10       contains: "SHGetSpecialFolderLocation"
11
12     - element: "number of arguments"
13       equals: 3
14
15     - element: "argument"
16       argument_index: 1
17       equals: 0x07

```

Listing 8.9: A Call Signature that matches a call to SHGetSpecialFolderLocation.

8.3 Service-based Persistence Techniques

In this section, we will discuss Call Signatures for [technique SP0](#). As we discussed in [subsection 3.3.3](#), there are multiple ways for an application to

⁷ https://docs.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetspecialfolderpathw

⁸ https://docs.microsoft.com/en-us/windows/win32/api/shlobj_core/nf-shlobj_core-shgetspecialfolderlocation

create a new service:

- Calling the `CreateService` API function.
- Using the `sc.exe` command-line application.
- Writing directly to the Windows Registry.

8.3.1 Creating Services Using the Windows API

To create a service that is automatically started using the Windows API, an application needs to call the `CreateService` function.

The fifth argument of `CreateService` is a constant that signifies how the new service should be started. As malware wants the service to be started automatically, it uses the `SERVICE_AUTO_START`⁹ constant. This constant has the value 2.

Listing 8.10 is a Call Signature that describes calls to `CreateService` with `SERVICE_AUTO_START` as the fifth argument.

```
1 ---
2
3 signature:
4     technique: "SP0"
5     description: >
6         This Call Signature can be used to
7         search for calls to CreateService.
8     rules:
9         - element: "function name"
10           contains: "CreateService"
11
12         - element: "number of arguments"
13           equals: 13
14
15         - element: "argument"
16           argument_index: 5
17           equals: 0x00000002
```

Listing 8.10: A Call Signature that matches a call to `CreateService`.

8.3.2 Creating Services Using Sc.exe

To create a service using the command line, malware needs to run the `Sc.exe` command. For example, the command in Listing 8.11

```
1 sc create ExampleService binpath="C:\temp\example.exe" start="
   auto" obj="LocalSystem"
```

Listing 8.11: An example usage of `sc.exe`.

⁹ <https://docs.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-createservice>

To find such commands in a binary, we need a Call Signature that has rules for the two strings that will always be a part of these commands: “sc” and “create”. The Call Signature in [Listing 8.12](#) has these rules.

```
1 ---
2
3 signature:
4     technique: "SP0"
5     description: >
6         This Call Signature can be used to
7         search for usage of sc.exe commands.
8     rules:
9         - element: "any argument"
10           contains: "sc"
11
12         - element: "any argument"
13           contains: "create"
```

Listing 8.12: A Call Signature that matches creating a service using the `sc.exe` application.

8.3.3 Creating Services Using the Registry

The configuration of services is stored in the Registry in the key `HKLM\SYSTEM\CurrentControlSet\Services`. It is possible to write to a subkey of this key directly, to create a new service.

To be able to detect this, we will use the same Call Signatures as in [section 8.1](#), but we will use `HKLM\SYSTEM\CurrentControlSet\Services` (instead of `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`).

8.4 Scheduled Tasks-based Persistence Techniques

In this section, we will discuss Call Signatures for [technique TP0](#). As we discussed in [subsection 3.3.3](#), there are multiple ways for an application to create a scheduled task:

- Directly calling Windows API functions.
- Using the `schtasks.exe` command-line application.
- Using the `at.exe` command-line application.

8.4.1 Creating Services Using the Windows API

To create a scheduled task using the Windows API directly, an application needs to interact with the Task Scheduler service. The Windows API provides two ways to achieve this. Both use the `CoCreateInstance` to create a handler for the Task Scheduler service.

The first method calls `CoCreateInstance` with the following constants as its arguments. These constants are GUIDs (discussed in [section 8.2](#)) The values of the first and fourth argument are:

- `CLSID_CTaskScheduler`: 148BD52A-A2AB-11CE-B11F-00AA00530503
- `IID_ITaskScheduler`: 148BD527-A2AB-11CE-B11F-00AA00530503

The second, newer method is similar to the first, but uses different constants:

- `CLSID_TaskScheduler`: 0F87369F-A4E5-4CFC-BD3E-73E6154572DD
- `IID_ITaskService`: 2FABA4C7-4DA9-4013-9697-20CC3FD40F85

[Listing 8.13](#) shows a Call Signature that has rules to constrain these constants.

```
1 ---
2
3 signature:
4     technique: "TP0"
5     description: >
6         This Call Signature can be used to
7         search for calls to CoCreateInstance.
8     rules:
9         - element: "function name"
10           equals: "CoCreateInstance"
11
12         - element: "number of arguments"
13           equals: 5
14
15         - element: "argument"
16           argument_index: 0
17           in:
18             - "2A D5 8B 14 AB A2 CE 11 B1 1F 00 AA 00 53 05 03"
19             - "9F 36 87 0F E5 A4 FC 4C BD 3E 73 E6 15 45 72 DD"
20           type: "bytes"
21
22         - element: "argument"
23           argument_index: 3
24           in:
25             - "27 D5 8B 14 AB A2 CE 11 B1 1F 00 AA 00 53 05 03"
26             - "C7 A4 AB 2F A9 4D 13 40 96 97 20 CC 3F D4 0F 85"
27           type: "bytes"
```

Listing 8.13: A Call Signature that matches a call to `CoCreateInstance`.

8.4.2 Creating Services Using `Schtasks.exe`

It is also possible to create a scheduled task using the command line, with the `schtasks.exe` application. For example, the command in [Listing 8.14](#).


```
1 schtasks /create /tn WindowsUpdate /tr "c:\windows\real_update.  
   exe /sc onidle /i 30
```

Listing 8.14: An example `schtasks.exe` command that runs a fictitious executable when a user is idle for 30 minutes.

Similar to `sc.exe` in [subsection 8.3.2](#), we need rules that capture the strings that will always be part of the command: “`schtasks`” and “`/create`”. Like the Call Signature in [Listing 8.15](#).

```
1 ---  
2  
3 signature:  
4     technique: "TP0"  
5     description: >  
6         This Call Signature can be used to  
7         search for usage of schtasks.exe commands.  
8     rules:  
9         - element: "any argument"  
10           contains: "schtasks"  
11  
12         - element: "any argument"  
13           contains: "/create"
```

Listing 8.15: A Call Signature that matches creating a scheduled task using the `schtasks.exe` application.

8.4.3 Creating Services Using `At.exe`

The Call Signature for `at.exe` is similar to the one for `schtasks.exe` because the applications themselves are similar. [Listing 8.16](#) shows an example of an `at.exe` command.

```
1 at 09:00 /interactive /every:m C:\Windows\System32\  
   WindowsUpdate\reverseshell.exe
```

Listing 8.16: An example `at.exe` command that runs a fictitious executable every Monday at 9:00.

To be able to find function calls that perform some operation on such commands, we need to find all occurrences of “`at`” and “`/every`”.

```

1 ---
2
3 signature:
4   technique: "TP0"
5   description: >
6     This Call Signature can be used to
7     search for usage of at.exe commands.
8   rules:
9     - element: "any argument"
10      contains: "at"
11
12     - element: "any argument"
13      contains: "/every:"

```

Listing 8.17: A Call Signature that matches creating a scheduled task using the `at.exe` application.

8.5 A Reflection on Writing Call Signatures

After writing the Call Signatures for this chapter, we wanted to share some thoughts on the process of writing Call Signatures.

Generally speaking, Call Signatures are quite expressive. We were able to express every constraint on function call elements we wanted, with a relatively small number of operators and types.

Especially only supporting three types (strings, numbers, and bytes) seemed to be enough to cover all situations we want. This is likely because many API functions use constants as arguments. Besides this, interesting arguments are often strings (e.g. Registry paths and commands for the command line).

In an earlier iteration of Call Signatures, we included the return type as a function call element. However, we noticed that we never used a constraint on the return type when writing Call Signatures. The return type was not needed, because the only times we know the return type is for Windows API functions, and we were already able to search for these using the function name. Writing rules that constraint the function name and arguments was enough for us.

In [section 8.2](#), we wrote Call Signatures for multiple similar API functions that are used to resolve paths. This resulted in multiple Call Signatures with only slight differences. It would be a good improvement to be able to combine these into one Call Signature.

Unfortunately, string parameters are not always passed directly into calls. Sometimes, they are first manipulated in some way. For example, they are turned into `CString` objects¹⁰ or formatted using `cprintf`¹¹. To

¹⁰<https://docs.microsoft.com/en-us/cpp/atl-mfc-shared/using-cstring>

¹¹<https://www.cplusplus.com/references/cstdio/printf/>

capture such situations, we wrote Call Signatures to search for function calls to unspecified functions with a string as one of their arguments. For example, the Call Signatures in [Listing 8.3](#) and [Listing 8.15](#). This allows us to detect many calls to functions that perform some action on a string of interest. This does however increase the chance of false positives.

Chapter 9

Experiments & Analysis

In [chapter 5](#), we presented Call Signatures, expressions to describe function calls of interest. We implemented an IDA Pro plugin, CSP, that searches for function calls using Call Signatures (in [chapter 6](#)). In [chapter 8](#), we used Call Signatures to describe persistence techniques (from [chapter 7](#)).

In this chapter, we will analyze whether CSP detects persistence techniques in known malware. In [section 9.1](#), we create datasets of real-world malware samples that implement each of the four techniques in [chapter 8](#). We run the CSP on each dataset with the relevant Call Signatures, to see for many samples it detects the persistence techniques.

To see how well CSP and the Call Signatures from [chapter 8](#) perform compared to existing tooling, we run Capa (earlier discussed in [chapter 2](#)) on the same datasets from [section 9.1](#). In [section 9.2](#), we analyze how Capa works, how it performs in our experiments, and why.

9.1 Detection of Persistence Techniques in Real-World Samples

To analyze the effectiveness of Call Signatures and CSP, we will run four experiments on real-world malware samples.

In [chapter 8](#), we wrote Call Signatures for four techniques (one of each category discussed in [chapter 7](#)): [technique RP0](#), [technique DP0](#), [technique SP0](#) and [technique TP0](#). For each of these techniques, we will create a dataset of samples that implement the technique and try to detect the technique in that dataset using CSP. The goal of these experiments is to see how many true positives and false negatives CSP (using the Call Signatures) finds in a dataset of true positives.

9.1.1 The Sources of Malware Samples

To create the datasets, we use the following (publicly available) sources for samples:

- APTMalware¹: A dataset of state-sponsored malware. This was created by Coen Boot for his master thesis [4].
- Capa Test Files²: Files used by Capa for testing.
- theZoo³: A public repository of live malware samples.

Together these datasets contain 3571 unique samples that range from simple banking malware to advanced state-sponsored malware samples.

9.1.2 Creating the Datasets

We use the three sources of samples in subsection 9.1.1 to create four datasets, one for each of the four techniques we will analyze.

To know which samples from subsection 9.1.1 implement each of the techniques, we used commercial dynamic analysis platforms (similar to Hybrid Analysis⁴, Intezer Analyze⁵, and the behavioral analysis of VirusTotal[20]). These dynamic analysis platforms run malware samples in a (secure and ephemeral) virtual environment. They report on the actions that the samples take (e.g. any IP address the malware tries to connect to) and record the changes in the operating system (e.g. any changed Windows Registry keys).

We used the results of the dynamic analysis of each sample to determine whether it implements technique RP0, technique DP0, technique SP0, or technique TP0. For example, if the dynamic analysis detected that a sample changed the Registry key HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run, we know that it implements technique RP0.

It is important to note that dynamic analysis is not a replacement for static analysis. We use it to establish a baseline of true positive samples, but it is not able to find all samples that implement each technique [10].

This resulted in the following datasets:

- Dataset RP: 162 samples that implement technique RP0. The hashes can be found in section C.1.
- Dataset DP: 134 samples that implement technique DP0. The hashes can be found in section C.2.

¹<https://github.com/cyber-research/APTMalware>

²<https://github.com/mandiant/capa-testfiles>

³<https://github.com/ytisf/theZoo>

⁴<https://www.hybrid-analysis.com>

⁵<https://analyze.intezer.com>

- Dataset SP: 133 samples that implement [technique SP0](#). The hashes can be found in [section C.3](#).
- Dataset TP: 28⁶ samples that implement [technique TP0](#). The hashes can be found in [section C.4](#).

9.1.3 Analyzing the Datasets

We analyzed each sample in each dataset with the Call Signatures written for the technique of the dataset. Specifically:

- We analyzed each sample in dataset RP with the Call Signatures from [section 8.1](#).
- We analyzed each sample in dataset DP with the Call Signatures from [section 8.2](#).
- We analyzed each sample in dataset SP with the Call Signatures from [section 8.3](#).
- We analyzed each sample in dataset TP with the Call Signatures from [section 8.4](#).

Because IDA Pro is designed to analyze a single executable at a time, we wrote a script that can batch process the samples. This script opens each sample in IDA Pro in headless mode (i.e. without a GUI), waits for the IDA Pro analysis to complete, runs the plugin, and stores the IDA Pro output in a log file. After a dataset is fully analyzed, we can analyze the logs of each sample to find how many persistence techniques were detected by CSP.

The average analysis time of a sample is twenty-one seconds. The majority of this time is taken up by the startup and analysis of IDA Pro and decompiling the functions.

9.1.4 Discussion of the Results

[Table 9.1](#) shows the results of this analysis. The first column shows the number of samples in each dataset. The second column shows how many samples were detected by CSP to implement the persistence technique in the dataset.

As we can see, CSP detected the persistence techniques in most samples. However, for three datasets, CSP did not detect the persistence technique in all samples. This is expected, as the datasets were made using dynamic analysis and the analysis was done using static analysis (we discussed the differences in [section 4.1](#)).

⁶We found significantly fewer samples for the TP dataset than for the other dataset. This might be a sign that scheduled task-based persistence techniques are less common than we thought.

Dataset	Total Samples	Detected by CSP
Dataset RP	162	115 (70.99%)
Dataset DP	134	134 (100%)
Dataset SP	133	129 (96.99%)
Dataset TP	28	19 (67.86%)

Table 9.1: The total number of samples and the number of detected samples by CSP in each dataset.

We can see the advantage of detecting functionality in binaries by looking at function calls in the results of [technique DP0](#) and [technique SP0](#). Both of these techniques are implemented using specific Windows API function calls (i.e. the functions used to resolve paths and `CreateService`), making them predictable and hard to hide (because regardless of the obfuscation, the malware will need to make the function call).

On the other hand, the [technique RP0](#) and [technique TP0](#) samples have a significantly lower detection percentage. We assume that this is because both techniques rely more heavily on strings, which are easier to obfuscate. Registry-based persistence techniques always use strings to specify the path of a Registry key and creating a Scheduled Task is more commonly done via the command line (i.e. also using strings), because it is easier to implement than using the Windows API (as we saw in [section 8.4](#)). This shows a fundamental weakness in CSP and Call Signatures (and static analysis in general): if data is obfuscated in a binary, it is impossible to know what a binary does, without first deobfuscating the data.

We manually analyzed about half of the false negatives (i.e. the samples that were not detected) and we found that the underlying reasons for the false negatives in each dataset fell into two categories⁷:

1. Limitations of IDA Pro: In some cases, IDA Pro was unable to decompile part of a binary. For example, a common limitation of IDA Pro is that it is unable to decompile exception handling⁸.
2. Obfuscation by the malware author: As discussed above, malware authors often employ obfuscation to hide data within the binary.
 - Some binaries placed data in a special resource section within the binary. When specific data is needed during runtime, it is dynamically resolved in the resource section. This makes it hard to know where specific data is used, without running the binary.
 - Some binaries encrypt strings by XORing each byte in a string and decrypting the strings at runtime.

⁷Analyzing the manual samples took us about three hours.

⁸<https://www.hex-rays.com/products/decompiler/manual/limit.shtml>

- Some binaries mix executable code and data throughout the binary, confusing the disassembler.
- Some binaries call offsets within functions to confuse the disassembler about what functions they call.

9.2 Comparison With Capa

In [section 9.1](#), we showed that CSP is capable of finding persistence techniques. In this section, we will compare CSP to Capa, to get an idea of how well CSP performs compared to existing tooling. As both Capa and CSP are static analysis tools that search for specific elements in a binary based on pre-defined conditions, the tools are easy to compare.

It is important to note that we are making this comparison to determine the effectiveness of Call Signatures and CSP, and not to showcase a replacement for Capa.

9.2.1 A Short Overview of Capa

Capa is a static malware analysis tool that detects the capabilities of malware samples. It is not limited to persistence but can detect many capabilities, ranging from simple interactions with the operating system (e.g. writing to a file) to more complex malware capabilities (e.g. communication with a C&C server). It does not use the disassembler of IDA Pro, but a standalone disassembler called Vivisect⁹.

Capa searches for specific elements (e.g. constants or Windows API function calls) in a binary. It pre-defines which elements it is looking for, for a specific capability, in *rules* (to prevent confusion with the rules of Call Signatures, we will refer to the rules in Capa as *Capa rules*). [Listing 9.1](#) shows an example of such Capa rule¹⁰.

Each rule consists of meta-information (e.g. the name and the author) and a list of elements that it is looking for. In [Listing 9.1](#), we see an example of a Capa rule.

⁹ <https://github.com/vivisect/vivisect>

¹⁰ <https://github.com/mandiant/capa-rules/blob/master/persistence/startup-folder/get-startup-folder.yml>


```

1 rule:
2   meta:
3     name: get startup folder
4     namespace: persistence/startup-folder
5     author: matthew.williams@mandiant.com
6     scope: basic block
7     att&ck:
8       - Persistence::Boot or Logon Autostart Execution::
        Registry Run Keys / Startup Folder [T1547.001]
9     examples:
10      - 07F7846BBCDA782E5639292AD93907EB:0x40121A
11   features:
12     - and:
13       - or:
14         - number: 0x07 = CSIDL_STARTUP
15         - number: 0x18 = CSIDL_COMMON_STARTUP
16       - or:
17         - api: shell32.SHGetFolderPath
18         - api: shell32.SHGetFolderLocation
19         - api: shell32.SHGetSpecialFolderPath
20         - api: shell32.SHGetSpecialFolderLocation

```

Listing 9.1: The get startup folder Capa rule.

Capa searches for the elements in a Capa rule in one of three scopes: a basic block, a function, or a binary. For example, in [Listing 9.1](#) all elements need to be present in a basic block. When all elements in a Capa rule are present in the scope of the Capa rule, Capa reports that a binary implements the capability that the rule describes.

In [Listing 9.1](#), we also see that it is possible to use conjunctions and disjunctions that allow users to define more complex expressions of elements (e.g. searching for 0x07 or 0x18). Because of this, Capa rules often define multiple ways to implement a technique (e.g. [Listing 9.1](#) defines multiple API functions that resolve paths) or multiple techniques (e.g. [Listing 9.1](#) is looking for both [technique DP0](#) and [technique DP1](#)).

For a detailed explanation of how Capa and its rules work, please read the documentation provided by Mandiant¹¹.

9.2.2 The Differences Between Call Signatures & Capa Rules

Call Signatures and CSP are similar to Capa. They both search for pre-defined patterns in a binary. However, they do have some nuanced, but fundamental differences. In our opinion, the three most important differences are:

1. The scope they operate on: Call Signatures and Capa rules are defined over a different scope. Call Signatures are defined over function calls,

¹¹<https://github.com/mandiant/capa-rules/blob/master/doc/format.md>

while Capa rules are defined over basic blocks, functions, or binaries (which can contain multiple function calls). Because Call Signatures use a smaller scope, we can say that Call Signatures are more precise (i.e. more restrictive) than Capa rules.

We discuss the practical implications of this difference in detail in [subsection 9.2.4](#).

2. Being able to detect indirect function calls: CSP is able to find indirect function calls (i.e. calls where the function address is computed at runtime), while Capa is not.

For example, if a malware sample implements [technique DP0](#) by calling `SHGetFolderPath` indirectly, the decompiled function call might look `?(?, 0x07, ?, ?, ?)`. Capa (using the Capa rule in [Listing 9.1](#)) will not detect this function call, because it does not see a call to `SHGetFolderPath`. However, CSP (using a Call Signature from [section 8.2](#)) is able to detect this call, because it does not require the function name to be known (discussed in [section 6.1](#)) and sees that the second argument is `0x07`.

3. The taxonomy of techniques: In [chapter 8](#), we use the clear taxonomy we laid out in [chapter 7](#) to write a Call Signature for each technique we want to be able to detect.

Capa does not provide such a taxonomy, making it unclear which Capa rules cover which techniques. This can result in Capa rules describing multiple techniques or techniques not being covered by any Capa rule.

[Listing 9.1](#) is a good example of one Capa rule covering multiple techniques. If the “get startup folder” Capa rule results in a match, we know that the malware uses a startup directory to achieve persistence. However, we do not know which directory, because the “get startup folder” Capa rule is used to detect both [technique DP0](#) and [technique DP1](#).

9.2.3 Experiments

To compare the detection performance of CSP (with Call Signatures) and Capa (with Capa rules), we run the same experiments from [section 9.1](#) using Capa:

- We try to detect [technique RP0](#) in dataset RP, using Capa and the “persist via Run registry key”¹² Capa rule¹³.

¹²Some Capa rules describe more than one persistence technique. For example, [Listing 9.1](#) describes both [technique DP0](#) and [technique DP1](#). To address this, we parse the output of Capa to make sure only the relevant elements are present.

¹³<https://github.com/mandiant/capa-rules/blob/master/persistence/registry/run/persist-via-run-registry-key.yml>

- We try to detect [technique DP0](#) in dataset DP, using Capa and the “get startup folder” Capa rule¹⁴.
- We try to detect [technique SP0](#) in dataset SP, using Capa and the “persist via Windows service” Capa rule¹⁵.
- We try to detect [technique TP0](#) in dataset TP, using Capa and the “schedule task via ITaskScheduler”¹⁶, “schedule task via command line”¹⁷, and “schedule task via ITaskService”¹⁸ Capa rules.

It is important to note that we are technically making two comparisons here: how well Call Signatures and Capa rules can describe persistence techniques and how well CSP and Capa can detect persistence techniques using Call Signatures and Capa rules, respectively.

[Table 9.2](#) shows the same table as [Table 9.1](#) with an additional column that shows the number of detected samples by Capa for each dataset.

Dataset	Total Samples	CSP	Capa
Dataset RP	162	115 (70.99%)	112 (69.14%)
Dataset DP	134	134 (100%)	134 (100%)
Dataset SP	133	129 (96.99%)	121 (90.98%)
Dataset TP	28	19 (67.86%)	18 (64.29%)

Table 9.2: The number of matches per dataset by CSP compared with the matches by Capa.

As we can see, CSP and Capa perform similarly, but Capa performs slightly worse on three of the four datasets. As Call Signatures are more precise than Capa rules (discussed in [subsection 9.2.2](#)), we would expect CSP to detect fewer samples than Capa. It is surprising to see that Capa detects slightly fewer samples than CSP.

Further analysis of the samples that were detected by CSP and not by Capa shows that the difference in matches can be explained by the second and third differences in [subsection 9.2.2](#) (i.e. CSP detecting indirect calls and differences in the rules):

- The Capa rule for [technique RP0](#) does not fully cover opening Registry keys from the command line (discussed in [subsection 3.3.2](#)).

¹⁴ <https://github.com/mandiant/capa-rules/blob/master/persistence/startup-folder/get-startup-folder.yml>

¹⁵ <https://github.com/mandiant/capa-rules/blob/master/persistence/service/persist-via-windows-service.yml>

¹⁶ <https://github.com/mandiant/capa-rules/blob/master/persistence/scheduled-tasks/schedule-task-via-itaskscheduler.yml>

¹⁷ <https://github.com/mandiant/capa-rules/blob/master/persistence/scheduled-tasks/schedule-task-via-command-line.yml>

¹⁸ <https://github.com/mandiant/capa-rules/blob/master/nursery/schedule-task-via-itaskservice.yml>

- The Capa rule for [technique SP0](#) does not fully cover creating Windows Services from the command line (discussed in [subsection 3.3.2](#)).
- Capa did not detect all calls to `CreateService` (the API function used in [technique SP0](#)), because the calls were made indirectly.
- The Capa rules for [technique TP0](#) do not cover creating a Scheduled Task from the command line via `at.exe` (discussed in [section 8.4](#)).
- Capa did not detect all calls to `CoCreateInstance` (the API function used in [technique TP0](#)), because the calls were made indirectly.

Most of these problems can be solved by adding new or modifying existing Capa rules and are not a fundamental limitation of Capa rules or Capa. However, this does show the importance of having a well-defined taxonomy of techniques and to not rely on function names.

9.2.4 False Positives

In the experiment in the previous section, we analyzed the false negatives of CSP and Capa. In this section, we discuss false positives.

Intuitively, Call Signatures and CSP are less prone to false positives than Capa, because Call Signatures operate on a smaller scope (as we discussed in the first difference in [subsection 9.2.2](#)). Capa leaves more room for false positives by looking for specific elements in basic blocks, functions or binaries regardless of how the elements are used and relate to each other.

Unfortunately, there is no automated way to know for certain that a sample does not implement a specific technique, which means that finding and analyzing false positives requires a lot of manual work.

During our research we noticed eleven samples that were detected (to implement [technique DP0](#)) by Capa, but not by CSP. Manual inspection revealed that these eleven samples do not implement [technique DP0](#). This shows that our intuition that Call Signatures and CSP are less prone to false positives than Capa is at least anecdotally true.

The decompiled function in [Listing 9.2¹⁹](#) shows us one of these false positives.

¹⁹The hash of the sample is 0102777ec0357655c4313419be3a15c4ca17c4f9cb4a440bfb16195239905ade.

```

1 int __usercall sub_1002A2E0@<eax>(int a1@<esi>)
2 {
3     WCHAR pszPath[622]; // [esp+Ch] [ebp-4ECh] BYREF
4     int v3; // [esp+4F4h] [ebp-4h]
5
6     *(_DWORD *)(a1 + 20) = 7;
7     *(_DWORD *)(a1 + 16) = 0;
8     *(_WORD *)a1 = 0;
9     v3 = 0;
10    memset(pszPath, 0, 0x26Du);
11    if ( SHGetFolderPathW(0, 28, 0, 0, pszPath) >= 0 )
12        sub_1000BC70(pszPath, a1, wcslen(pszPath));
13    return a1;
14 }

```

Listing 9.2: Pseudo code of a call to `SHGetFolderPath`, decompiled by IDA Pro.

To detect [technique DP0](#), Capa searches for two elements: a call to `SHGetFolderPath` and the constant 7 (representing the user startup directory). We can see that `SHGetFolderPath` is called in [Listing 9.2](#) (on line 11), but it is used to resolve the path to the Application Data directory (represented by the constant 28). Capa also finds the constant 7 in the function (on line 6), but it is unrelated to the call to `SHGetFolderPath`.

CSP does not detect [technique DP0](#) in [Listing 9.2](#) (and neither does it for the other ten cases), because it looks at a smaller scope (i.e. the function call to `SHGetFolderPath`) and it does not find a function call that looks like `SHGetFolderPath(?, 7, ?, ?, ?)`. This shows that tools have a larger scope (like Capa) can be more prone to false positives than tools that have a smaller scope (like Call Signatures and CSP).

Chapter 10

Future Work

- **Other Functionality & Architectures** The scope of this thesis is detection of persistence in 32-bit Windows malware. An interesting continuation of this research would be to see if the results hold if we broaden the scope.

There are multiple options for broadening the scope:

- **High-level functionality:** We chose to focus on persistence, but, as discussed in [chapter 4](#), persistence is only one part of the malware life cycle. The same detection can be used for other high-level functionality (e.g. lateral movement or C&C communication).
- **Low-level functionality** It would be interesting to research whether it is possible to detect low-level functionality by searching for function calls. Interesting low-level functionality to look at includes cryptography and networking.
- **Architectures:** We chose to focus on 32-bit Windows binaries, as most malware is Windows-based and compiled for 32-bit Windows. However, the detection technique based on function calls we present in [chapter 5](#), can also be applied to different binary formats (e.g. ELF), architectures (e.g. ARM), and 64-bit versions of operating systems.

A logical continuation of this research would be to focus on 64-bit Windows as malware is most prevalent on Windows and some malware is made specifically for 64-bit Windows [12]. However, the calling conventions used in 64-bit Windows are more complex than those used in 32-bit Windows, making detection of how many and which arguments are passed during a call harder.

Using IDA Pro will be useful, as it has built-in support for many CPU architectures (as discussed in [subsection 4.1.2](#)), including x86-64.

- **Countermeasures** A logical continuation of research about detection is to look at how to circumvent the detection. This research could focus on bypassing detection of a specific (persistence) technique or bypassing detection based on function calls in general (e.g. by rewriting `call` instructions to `jmp` instructions).
- **Taxonomy** In [chapter 7](#), we provide a taxonomy of persistence techniques. We used our own classifications, because the existing taxonomy for malware capabilities, MITRE ATT&CK, is not as fine-grained as we need. Research into creating a model for a fine-grained taxonomy, would be a good contribution. Such a taxonomy could have the following layers:
 1. The high-level functionality (e.g. persistence): What malware is trying to achieve in broad terms.
 2. The low-level functionality (e.g. writing to a specific Registry key): The techniques used to achieve the high-level functionality.
 3. The implementation (e.g. using the `CreateRegKey` API function): The ways in which malware can implement low-level functionality. For example, in [subsection 3.3.2](#), we discussed multiple ways to interact with the Windows Registry.
- **Compatibility with Dynamic Analysis** This thesis is about static analysis. However, malware analysis does not have to choose between static and dynamic analysis. It is often useful to use both complementary. It would be interesting to look at how our methodology can be used together with dynamic analysis tools (such as debuggers). For example, `Funcap`¹ is a tool that records the arguments of function calls it sees while a binary is running. Another possibility would be automatically setting breakpoints at function calls detected by CSP.
- **General Applications of Call Signature Detection** Call Signatures and CSP allow us to search for function calls in binaries. These function calls do not have to be used in malware functionality and the binaries do not have to be malicious. Searching for function calls might also be useful in other applications. For example, code re-use detection or malware detection.

¹<https://github.com/deresz/funcap>

Chapter 11

Conclusions

In malware analysis, it is important to better understand the functionality of a binary, because the source code of malware is generally not available and malware employs obfuscation techniques to hide its functionality. As executable binaries are complex data structures, it is often necessary to understand the instructions and control flow in a binary, and not only look at the binary as byte sequences, as we saw in the experiment in [Appendix A](#).

In this thesis, we present a method to detect high-level functionality in malware binaries. This method consists of two steps: (1) describing the function calls that are used to implement a capability as patterns, including the arguments that are passed to such functions and (2) searching for these patterns in malware binaries.

In [chapter 5](#), we introduced *Call Signatures*, patterns on the function name, the number of arguments and the arguments of a function call.

In [chapter 6](#), we developed an IDA Pro plugin, called *CSP*, that uses Call Signatures to search for function calls in a binary. Implementing the logic to search for function calls that match Call Signatures in an IDA Pro plugin has the advantage of being able to take advantage of the state-of-the-art disassembler and decompiler of IDA Pro.

To hinder malware analysis, malware authors often obfuscate the functionality of their malware. However, our approach focuses on Windows API function calls, which are well-defined and the arguments that are passed to them are predictable. This makes API calls harder to obfuscate and better for pattern matching. Even if a function call is made indirectly (i.e. the address and function name are unknown), we can still match the function call based on the arguments it is given.

To showcase the usage and effectiveness of Call Signatures and CSP, we used them to detect persistence techniques in 32-bit Windows malware samples. We chose persistence as it is commonly implemented by malware. We chose 32-bit Windows as most malware is Windows-based and compiled for 32-bit Windows [\[35\]](#) [\[12\]](#).

As existing classifications of malware functionality (such as MITRE ATT&CK) are not fine-grained enough, we provided our own taxonomy of common persistence techniques (in [chapter 7](#)). We found that persistence techniques generally fall into four categories (based on which Windows feature they rely on): registry-based, directory-based, service-based, and scheduled task-based techniques

For one technique of each category, we wrote Call Signatures ([chapter 8](#)) and created a dataset of malware samples that implement the technique ([chapter 9](#)). The SHA-256 hashes of each dataset are available in [Appendix C](#). CSP (using the Call Signatures) was able to detect the technique in most samples in each dataset. The lowest detection rate of a dataset is 67.86% and the highest is 100%. The two main reasons CSP was unable to detect a persistence technique are obfuscation by the malware and limitations in the decompiler of IDA Pro.

Obfuscation is a limitation of all static analysis tools: if a malware sample obfuscates the arguments it uses (e.g. by encrypting them), static analysis will not be able to detect them.

We also ran a tool similar to our research, Capa, on each dataset ([section 9.2](#)), to see how well Call Signatures and CSP perform, compared to existing tooling. We found that Call Signatures and CSP were able to perform as well as (and in some cases, slightly better than) Capa. We found that CSP detected more samples than Capa than our Call Signatures describing more ways to implement a technique and CSP being able to detect function calls without knowing the function name. Finally, we analyzed (in [subsection 9.2.4](#)) why Call Signatures and CSP can be less prone to false positives than Capa.

In summary, we showed that function calls can be a good indicator of high-level malware functionality and searching for patterns on a function call level can be beneficial over or alongside existing detection methods. We provided Call Signatures and CSP to search for function calls in binaries.

Bibliography

- [1] 0x6d696368. *Ghidra FID generation*. <https://blog.threatrack.de/2019/09/20/ghidra-fid-generator/>. Sept. 20, 2019.
- [2] AV-TEST. <https://www.av-test.org/en/statistics/malware/>. Mar. 16, 2022.
- [3] Chris Bisnett and Kyle Hanslovan. “Evading Autoruns”. DerbyCon 7.0 <https://youtube.com/watch?v=AEmuhCwFL5I>. 2017.
- [4] Coen Boot. “Applying Supervised Learning on Malware Authorship Attribution”. Master Thesis. Radboud University, 2019. URL: https://www.ru.nl/publish/pages/769526/b_coen_boot.pdf.
- [5] Chainalysis. *North Korean Hackers Have Prolific Year as Their Unlaundered Cryptocurrency Holdings Reach All-time High*. <https://blog.chainalysis.com/reports/north-korean-hackers-have-prolific-year-as-their-total-unlaundered-cryptocurrency-holdings-reach-all-time-high/>. Jan. 13, 2022.
- [6] Raj Chandel. *Windows Persistence using WinLogon*. <https://www.hackingarticles.in/windows-persistence-using-winlogon/>. Apr. 12, 2020.
- [7] Chris Eagle. *The IDA Pro Book: The Unofficial guide to the World’s most popular disassembler*. No Starch Press, 2011.
- [8] Najla Etaher, George R.S. Weir, and Mamoun Alazab. “From Zeus to Zitmo: Trends in Banking Malware”. In: *2015 IEEE Trustcom/Big-DataSE/ISPA*. Vol. 1. 2015. DOI: [10.1109/Trustcom.2015.535](https://doi.org/10.1109/Trustcom.2015.535).
- [9] Andrea Fortuna. *Malware persistence techniques*. <https://andreafortuna.org/2017/07/06/malware-persistence-techniques/>. July 6, 2017.
- [10] Yuxin Gao, Zexin Lu, and Yuqing Luo. “Survey on malware anti-analysis”. In: *Fifth International Conference on Intelligent Control and Information Processing*. 2014.
- [11] Andy Greenberg. *Sandworm: A new era of cyberwar and the hunt for the Kremlin’s most dangerous hackers*. Doubleday, 2019.

- [12] Dalya Guttman. *Beware of the 64-bit Malware*. Tech. rep. <https://www.deepinstinct.com/blog/whitepaper-beware-of-the-64-bit-malw/>. Deep Instinct, May 2017.
- [13] Irfan Ul Haq and Juan Caballero. “A Survey of Binary Code Similarity”. In: *ACM Computing Surveys* 54.3 (Apr. 17, 2021).
- [14] HexRays. *F.L.I.R.T.* <https://hex-rays.com/products/ida/tech/flirt/>. 2021.
- [15] Eric Hutchins, Michael Cloppert, Rohan Amin, et al. “Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains”. In: *Leading Issues in Information Warfare & Security Research* 1.1 (2011).
- [16] Jesse Kornblum. “Identifying almost identical files using context triggered piecewise hashing”. In: *Digital Investigation* 3 (Sept. 1, 2006).
- [17] Sushil Kumar and Sudhakar. “An emerging threat Fileless malware: a survey and research challenges”. In: *Cybersecurity* 3.1 (2020).
- [18] Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. “Function matching-based binary-level software similarity calculation”. In: *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. Association for Computing Machinery, Oct. 1, 2013.
- [19] Anartz Martin. *Attack detection fundamentals: Code execution and persistence lab 2*. <https://labs.f-secure.com/blog/attack-detection-fundamentals-code-execution-and-persistence-lab-2/>. July 3, 2020.
- [20] Emiliano Martinez. *VirusTotal += Behavioural Information*. <https://blog.virustotal.com/2012/07/virustotal-behavioural-information.html>. July 23, 2012.
- [21] Luca Massarelli et al. “SAFE: Self-Attentive Function Embeddings for Binary Similarity”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. 2019.
- [22] Microsoft. *Defining Malware*. [https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948\(v=technet.10\)](https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10)). Jan. 4, 2009.
- [23] Microsoft. *Load-Time Dynamic Linking*. <https://docs.microsoft.com/en-us/windows/win32/dlls/load-time-dynamic-linking>. July 1, 2021.
- [24] Microsoft. *Run and RunOnce Registry Keys*. <https://docs.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys>. Feb. 8, 2022.

- [25] Microsoft. *Running 32-bit Applications*. <https://docs.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications>. 2020.
- [26] Jiang Ming et al. “BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Aug. 2017.
- [27] Nitin Naik et al. “Fuzzy-Import Hashing: A Malware Analysis Approach”. In: *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. ISSN: 1558-4739. July 2020.
- [28] Anitta Patience Namanya et al. “Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing”. In: *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*. Aug. 2016.
- [29] Edward Raff et al. “Malware Detection by Eating a Whole EXE”. In: *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*. June 20, 2018.
- [30] Bander Al-rimy, Mohd Maarof, and Syed Shaid. “Ransomware threat success factors, taxonomy, and countermeasures: A survey and research directions”. In: *Computers & Security* 74 (2018).
- [31] David E. Sanger. *The perfect weapon: War, sabotage, and fear in the Cyber Age*. Broadway Books, 2019.
- [32] Ali Aydın Selçuk, Fatih Orhan, and Berker Batur. “Undecidable Problems in Malware Analysis”. In: *2017 12th International Conference for Internet Technology and Secured Transactions*. 2017.
- [33] Agam Shah. *Apple is beginning to undo decades of Intel, x86 dominance in PC market*. https://www.theregister.com/2021/11/12/apple_arm_m1_intel_x86_market/. Nov. 12, 2021.
- [34] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. <https://nostarch.com/malware>. No Starch Press, 2012.
- [35] AV-TEST. *Security Report 2019/2020*. Tech. rep. <https://www.av-test.org/en/news/facts-analyses-on-the-threat-scenario-the-av-test-security-report-2019-2020/>. AV-TEST, Aug. 2020.
- [36] Alan Turing. “On Computable Numbers, with an application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42.2 (Nov. 12, 1936).
- [37] William Turton and Kartikay Mehorta. *Hackers Breached Colonial Pipeline Using Compromised Password*. <https://www.bloomberg.com/news/articles/2021-06-04/hackers-breached-colonial-pipeline-using-compromised-password>. June 4, 2021.

- [38] Daniel Uroz and Ricardo Rodríguez. “Characteristics and detectability of windows auto-start extensibility points in memory forensics”. In: *Digital Investigation* 28 (Apr. 1, 2019).
- [39] Angel M. Villegas. “Function identification and recovery signature tool”. In: *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, Oct. 2016.
- [40] Xiaojun Xu et al. “Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Oct. 30, 2017.
- [41] Kim Zetter. *Countdown to Zero Day: Stuxnet and the launch of the world’s first Digital Weapon*. Crown Archetype, 2015.

Appendix A

Finding Common Traits Using Binlex

At the beginning of this research, we tried to detect persistence in malware binaries with Binlex¹, a static analysis tool that interprets binaries as a sequence of bytes. This experiment taught us that we need to look more semantically at the instructions in binaries to detect their functionality.

Binlex extracts *traits* from binaries. A trait is a function or a basic block, with the memory operands of each instruction removed. For example, the two instructions shown in Listing A.1 result in the trait “33 c0 ff 15 ?? ?? ??”. As we can see, the trait contains the same bytes as the original basic block, but the address operand of the `call` instruction is replaced by wildcard characters.

```
1 33 c0 xor eax, eax
2 ff 15 d8 60 40 00 call 0x4060d8
```

Listing A.1: An example of a basic block made up of two instructions.

We wanted to see whether it is possible to use Binlex to find common traits in two samples that implement the same persistence technique. If this is possible, we can search for the traits relevant to a persistence technique in other malware binaries to detect the technique.

We found two malware samples that implement technique RP3. Their SHA-256 hashes are available in section C.5.

We manually analyzed `Lab06-03.exe` in IDA Pro to understand how it implements the persistence technique. Listing A.2 shows the basic block that executes the Windows API functions `RegOpenKeyEx` and `RegSetValueEx` to write to the registry for persistence.

¹<https://github.com/c3rb3ru5d3d53c/binlex>

```

1 lea     ecx, [ebp+phkResult]
2 push    ecx           ; phkResult
3 push    0F003Fh       ; samDesired
4 push    0             ; ulOptions
5 push    offset SubKey  ; "Software\\Microsoft\\Windows
   "...
6 push    80000002h     ; hKey
7 call    ds:RegOpenKeyExA
8 push    0Fh          ; cbData
9 push    offset Data    ; "C:\\Temp\\cc.exe"
10 push    1            ; dwType
11 push    0            ; Reserved
12 push    offset ValueName ; "Malware"
13 mov     edx, [ebp+phkResult]
14 push    edx           ; hKey
15 call    ds:RegSetValueExA
16 test    eax, eax
17 jz      short loc_4011D2

```

Listing A.2: A basic block, disassembled by IDA Pro, that implements [technique RP3](#).

[Listing A.3](#) shows the Binlex trait that corresponds to the basic block in [Listing A.2](#).

```

1      8d 4d ??
2      51
3      68 3f 00 0f 00
4      6a 00
5      68 70 71 40 00
6      68 02 00 00 80
7      ff 15 ?? ?? ?? ??
8      6a 0f
9      68 a0 71 40 00
10     6a 01
11     6a 00
12     68 68 71 40 00
13     8b 55 ??
14     52
15     ff 15 00 ?? ?? ??
16     85 c0
17     74 0d

```

Listing A.3: The Binlex trait corresponding to the assembly in [Listing A.2](#).

Our goal was to find a trait that is similar to [Listing A.3](#) in the other binary. Unfortunately, there are only 29 common traits, and they are small (7 bytes at most). None of them resembled (or were part of) [Listing A.2](#). This tells us that the common traits have nothing to do with the persistence technique, but are either common in general or are coincidentally in both binaries.

Even though the samples implement [technique RP3](#) similarly (e.g. they both use `RegOpenKeyEx` and `RegSetValueEx` without any obfuscation), their

implementations differ too much to find any common basic blocks. If such similar implementations do not have any common traits, binaries that use vastly different implementations (e.g. by using different API calls or a more complex call structure) will not have any common traits either. This shows us that purely looking at byte sequences will not capture the relationships in the binaries we are looking for.

Although Binlex does not seem to be the right tool for our research purposes, it is a valuable tool in detecting code re-use in (malware) binaries. If two malware binaries have many common (larger) traits, it is a good indication that they share code. This is valuable during analysis, as it links the two malware samples (e.g. it may indicate that they have been written by the same author).

Appendix B

Example Code Used in the Dynamic Linking Comparison

This appendix contains two semantically equivalent code examples that differ in the way they resolve Windows API functions. These are used in [section 3.3.1](#).

```
1  #include <iostream>
2  #include <Tchar.h>
3  #include <windows.h>
4
5  int main()
6  {
7      const LPCWSTR KEY_NAME = _T(
8          "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"
9      );
10     const LPCWSTR VALUE_NAME = _T("ProductName");
11
12     ULONG result;
13
14     HKEY handle;
15     WCHAR buffer[128];
16     ULONG bufferSize = sizeof(buffer);
17
18     result = RegOpenKeyEx(
19         HKEY_LOCAL_MACHINE,
20         KEY_NAME,
21         NULL,
22         KEY_READ | KEY_WOW64_64KEY,
23         &handle
24     );
25     if (result != ERROR_SUCCESS) {
26         return result;
27     }
28 }
```

```

29     result = RegQueryValueEx(
30         handle,
31         VALUE_NAME,
32         NULL,
33         NULL,
34         (LPBYTE)buffer,
35         &bufferSize
36     );
37     if (result != ERROR_SUCCESS) {
38         return result;
39     }
40
41     result = RegCloseKey(handle);
42     if (result != ERROR_SUCCESS) {
43         return result;
44     }
45
46     std::wcout << "'" << buffer << "'" << "\n";
47
48     return 0;
49 }

```

Listing B.1: Example code that uses load-time dynamically linked function calls.

```

1  #include <iostream>
2  #include <Tchar.h>
3  #include <windows.h>
4
5  int main()
6  {
7      typedef LSTATUS(WINAPI *fOpenKey)(
8  HKEY, LPCWSTR, DWORD, REGSAM, PHKEY
9      );
10     typedef LSTATUS(WINAPI *fQueryKey)(
11         HKEY, LPCWSTR, LPDWORD, LPDWORD, LPBYTE, LPDWORD
12     );
13     typedef LSTATUS(WINAPI *fCloseKey)(HKEY);
14
15     const LPCWSTR KEY_NAME = _T(
16         "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"
17     );
18     const LPCWSTR VALUE_NAME = _T("ProductName");
19
20     ULONG result;
21
22     HKEY handle;
23     WCHAR buffer[128];
24     ULONG bufferSize = sizeof(buffer);
25
26     HMODULE hLib = LoadLibrary(_T("Advapi32.dll"));
27     if (hLib == NULL) {
28         return GetLastError();
29     }

```

```

30
31     fOpenKey hRegOpenKeyEx = (fOpenKey) GetProcAddress(
32         hLib, "RegOpenKeyExW"
33     );
34     fQueryKey hRegQueryValueEx = (fQueryKey) GetProcAddress(
35         hLib, "RegQueryValueExW"
36     );
37     fCloseKey hRegCloseKey = (fCloseKey) GetProcAddress(
38         hLib, "RegCloseKey"
39     );
40     if (hRegOpenKeyEx == NULL ||
41         hRegQueryValueEx == NULL ||
42         hRegCloseKey == NULL) {
43         return GetLastError();
44     }
45
46     result = hRegOpenKeyEx(
47         HKEY_LOCAL_MACHINE,
48         KEY_NAME,
49         NULL,
50         KEY_READ | KEY_WOW64_64KEY,
51         &handle
52     );
53     if (result != ERROR_SUCCESS) {
54         return result;
55     }
56
57     result = hRegQueryValueEx(
58         handle,
59         VALUE_NAME,
60         NULL,
61         NULL,
62         (LPBYTE)buffer,
63         &bufferSize
64     );
65     if (result != ERROR_SUCCESS) {
66         return result;
67     }
68
69     result = hRegCloseKey(handle);
70     if (result != ERROR_SUCCESS) {
71         return result;
72     }
73
74     FreeLibrary(hLib);
75
76     std::wcout << "'" << buffer << "'" << "\n";
77
78     return 0;
79 }
80

```

Listing B.2: Example code that uses run-time dynamically linked function calls.

Appendix C

Dataset Hashes

This appendix contains the SHA-256 hashes of the malware samples used in the datasets used in the experiments in [chapter 9](#).

All these samples come from APTMalware, the Capa Test Files or theZoo (see [subsection 9.1.1](#) for more information).

C.1 The Hashes of Dataset RP

- 0007ccdddb12491e14c64317f314c15e0628c666b619b10aed199eefcfe09705
- 00f60edc9acb15a56d49296418a018da4fd7477315e943a8eed26f8c3b6e8651
- 017f4349170bd50e0abe565cd96ce7c65cf9a8308f76a20a0a7f391f73390012
- 048329f3ac884ee53ff6e7509fedce7e5f44939fe509e5fdda7b323db0990166
- 066346170856972f6769705bc6ff4ad21e88d2658b4cacea6f94564f1856ed18
- 0b74282d9c03affb25bbecf28d5155c582e246f0ce21be27b75504f1779707f5
- 0c50ddf7295d4ddfafae479e7c3ce21ca6416442c0c8c5e90aedbb3e583a8b20
- 0c9b20f4cb0b3206f81c2afbb2ee4d995c28f74f38216f7d35454af624af8876
- 0e829513658a891006163ccbf24efc292e42cc291af85b957c1603733f0c99d4
- 0f10ddc73d739033e7a4731936610f60102b8f97cf3a66620b326b7a59fb6ff4
- 0fbb47373b8bbeffdf9377dc26b6418d2738e6f688562885f4d2a1a049e4948e
- 11fab8361a942e46375bd5ac259146fda20608594e265bcc1d3c011ab4c17226
- 139f7a50e8ff402759a3c118bbcc700732d6e8b63b55a7fb31d12030722ca5a4
- 13e40ee7c6874e2f1ed58bc09738a5525f86361f1a85387b2110f114d8f3272a
- 1419ba36aae1daecc7a81a2dfb96631537365a5b34247533d59a70c1c9f58da2
- 180e5227aae20fa2d6ae421835dc7d92f9393681c3006213dc2f6e3fbd07e3de
- 1a4a64f01b101c16e8b5928b52231211e744e695f125e056ef7a9412da04bb91
- 1acae7c11fb559b81df5fc6d0df0fe502e87f674ca9f4aefc2d7d8f828ba7f5c
- 1ba4f8d569dafdf2c0152d706fc9cc3d6eb646e8ea639c410c8f95e07bc2551e
- 1ba99d553582cc6b6256276a35c2e996e83e11b39665523f0d798beb91392c90
- 1bc9ab02d06ee26a82b5bd910cf63c07b52dc83c4ab9840f83c1e8be384b9254
- 1ef47da67f783f8cc8cda7481769647b754874c91e0c666f741611dec878c19

- 23f28b5c4e94d0ad86341c0b9054f197c63389133fcd81dd5e0cf59f774ce54b
- 25cbd45f2510444f86b10507e2884888decee0a5bec4bbab073cc6a6840b3a86
- 269ea4b883de65f235a04441144519cf6cac80ef666eccf073eedd5f9319be0f
- 28ce6aa6b39743680b5fe34bcd9822db3d30716cc3faf09bf37d0bc05a73d9e
- 2965c1b6ab9d1601752cb4aa26d64a444b0a535b1a190a70d5ce935be3f91699
- 2b160b7eef5ce5fdb83889f96fc40cbbbc7b85450ff2afdf781a8eb5d6a0f541
- 31488f632f5f7d3ec0ea82eab1f9baba16826967c3a6fa141069ef5453b1eb95
- 32144ba8370826e069e5f1b6745a3625d10f50a809f3f2a72c4c7644ed0cab03
- 32edd18cc8c458186b76cfa546fe7a394de3c48366ea4854e6f6a75727026780
- 3577845d71ae995762d4a8f43b21ada49d809f95c127b770aff00ae0b64264a3
- 369b9dcfa2430d63e91b03f5d1330e67e78ae3fd197f64f6518a1b4191c08cad
- 36f53eb7ea9e049ab589b69f748e4098be5f51577c3b881cf95d7928d19fe68d
- 37f4e9d0153221d9a236f299151c9f6911a6f78fff54c91b94ea64d1f3a8872b
- 387d4ea82c51ecda162a3ffd68a3aca5a21a20a46dc08a0ebe51b03b7984abe9
- 388ef4b4e12a04eab451bd6393860b8d12948f2bce12e5c9022996a9167f4972
- 399e19d2dc5e96531666a8cc3071115cf9b19ba1d1676294ee77bd2c75d25add
- 3cd42e665e21ed4815af6f983452cbe7a4f2ac99f9ea71af4480a9ebff5aa048
- 3de4f547b6ef69c9d60c1670d9dc93807eafeb15ffcf510fb1142b552b7214e9
- 402a8e7c29135edeed5936c7b5d3524f095bdab37658999fc3fa636b6b38e027
- 41dd95533d85a0fd099ee79fbb4c8699ae6f9299b74034b8bafa3b0ea4a1fb3a
- 43608e60883304c1ea389c7bad244b86ff5ecf169c3b5bca517a6e7125325c7b
- 466bb7da773c7c200f87a8a06f143c6c6856e9ebc4347eb4afb096104bcd97b4
- 468bef292236e98a053333983f7094f64551a05509837c775fa65fdb785ca95a
- 48f0bbc3b679aac6b1a71c06f19bb182123e74df8bb0b6b04ebe99100c57a41e
- 4b547b3992838cfb3b61cb25f059c0b56c2f7caaa3b894dbc20bf7b33dad5a1
- 4fe6757544c655c0228ca9fbcd3e197cd921e1e12776f5ad391cb9ac32d3667
- 5412cddde0a2f2d78ec9de0f9a02ac2b22882543c9f15724ebe14b3a0bf8cbda
- 5475ae24c4eeadcbd49fcd891ce64d0fe5d9738f1c10ba2ac7e6235da97d3926
- 593849098bd288b7bed9646e877fa0448dcb25ef5b4482291fdf7123de867911
- 5961861d2b9f50d05055814e6bfd1c6291b30719f8a4d02d4cf80c2e87753fa1
- 5bf2dfcf19db065cff2d55a9942c8fc8d5cbf77b58051ebf68ec6343cad91c16
- 5f23a3442fa4515ebba8e24f2254b52b3e4b000f12843a4f612da65de38db1de
- 5ffa6550de0191e329511c3251a04f13658ca783fb4bd72b2f11ad0271dfa376
- 6367cb0663c2898aff64440176b409c1389ca7834e752b350a87748bef3a878b
- 646194791590993c21a49e16465c245094e288c077d1e279258c3d22de0feb78
- 674ba1d7103cab6082ac34940962711b1f5bb7ae152b81051aa92ed6b9d6326e
- 677cbeea7c87e4e03da87d71137897b200e2b0170950ddc958a72c09674b1685
- 67ecc3b8c6057090c7982883e8d9d0389a8a8f6e8b00f9e9b73c45b008241322
- 69b555a37e919c3e6c24cfe183952cdb695255f9458b25d00d15e204d96c737b
- 6a5a9b0ae10ce6a0d5e1f7d21d8ea87894d62d0cda00db005d8d0de17cae7743

- 6bb937d1a3d6fdce5108af79716b20b0c8a609763f10135d9a62314ced2a5fb8
- 6c0c3172ba927af3b3cb7a8dec90ba35a1e1d4f0b9086e1385de83c682860f73
- 6e5f4296bffa7128b6e8fa72ad1924d2ff19b9d64775bd1e0a9ce9c5944bd419
- 70992a72412c5d62d003a29c3967fcb0687189d3290ebbc8671fa630829f6694
- 722b179b39fc0a4556b2d92d4066a17b5a3e7dde0506a5c51c9304d1ba11118f
- 72a8fa454f428587d210cba0e74735381cd0332f3bdcbb45eecb7e271e138501
- 72e01e875b93e6808e8fff0e8a8f19b842ed213a9fcb38c175f6e8533af57d51
- 73794263b657632805c8c3907e2f20a9743d8c9b83aa3e21629ecce5de02b1ca
- 74e348068f8851fec1b3de54550fe09d07fb85b7481ca6b61404823b473885bb
- 7bf330666c586e8adf6751c911a2821fe67d39f34c8f10d3618fddcbd81a9e53
- 7cdb9c2e8b6ca7f0a683a39c0bdadc7a512cff5d8264fdec012c541fd19c0522
- 7d39126d6ed6dac39a23dcc0b9df6d0ae5124bb443489ab6e889f49cc1012e31
- 7d4e44037e53b6e5de45deb9ee4cf5921b52f8eb1073136f7c853e6f42516247
- 7dd063acdfb00509b3b06718b39ae53e2ff2fc080094145ce138abb1f2253de4
- 7f489a9493c233018342defc7e9140e1c8b6ecdc9d0baa31c9c0ee62c844272d
- 801a2d3687e64f6d0f70fefe1ac09686716e7317b88f3b24e54e76957601346f
- 81e5e73452aa8b14f6c6371af2dccab720a32fadfc032b3c8d96f9cdaab9e9df
- 8531447784b5082e45d3f61385fc6355b27dfd7ae354f4bf14db3aecf2fa1cce
- 85bca7ec25d2b226be5e8e20d51fc7718366f70291a398b6aa617d734a3e7ba1
- 86f5f5e5ea9bdbfb8b139cd9bc22826cea431f347f54035c5bc7a3f315d5f2f7
- 89e0016fc5bd3cd4e25f88c70f9f8f13f81a45e3c6dc8ac2a4be44b5c5274957
- 8c414cc53009cef9ad9aa3ffd766085a7b76aa56f69a12c3231268cc4b1d81
- 8cc3ede145613b926268828965830ad7fbcf0b6db2b8772b4d485a55b88dd308
- 8e222cb1a831c407a3f6c7863f3faa6358b424e70a041c196e91fb7989735b68
- 8f939e65e9ffedd16ae86687e154adbe607d56950d082778300039283f2f8330
- 910f55e1c4e75696405e158e40b55238d767730c60119539b644ef3e6bc32a5d
- 913c21141966750cfe80d1f64f7c819ae59e401b47f0b5031fd2486c10403c91
- 92c959c36617445a35e6f4f2ee2733861aa1b3baf8728d19a4fd5176f3c80401
- 92dbbe0eff3fe0082c3485b99e6a949d9c3747afa493a0a1e336829a7c1faafb
- 94200da50341169c8d6ada187b10adb6ae4d8eb5ebf1489b91b296b83545fe7c
- 95b6e427883f402db73234b84a84015ad7f3456801cb9bb19df4b11739ea646d
- 97187a61b57d238bc7fd0092d570c5ab0cfcc132cf3b0969e2f6e4190b1fa942
- 97ab07c8020aead6ce0d9196e03d3917045e65e8c65e52a16ec6ef660dd96968
- 98bd5e8353bc9b70f8a52786365bcd8b28bd3aef164d62c38dae8df33e04ac11a
- 9ad81024edaf1cbc28caa02051ab6ab6b7ca929cffa8a465d1bdaa452cf0b8f5
- 9cc38ea106efd5c8e98c2e8faf97c818171c52fa3afa0c4c8f376430fa556066
- 9d483ce0d62d78bf7692608bbcb25026257e6d667273481ca2ffe1641a64a9b4
- a0a422d24a3d60e47916d6bc6618924cbd69d815b4cb79cd3d120e2d2ccc90c4
- a0c13be0775dc91b1d0b80b64861d9ce491765565758a4bc47b67f4b3c3ffdd4
- a3a6f0dc5558eb93afa98434020a8642f7b29c41d35fa34809d6801d99d8c4f3

- a5373b33ac970dedeb52528b123959145bf51c95b159a30a7823ad8018ac4b41
- a747aab7d9fa397b28db3b7fbec771ce9b296fc701cc98be6b2d0e44364ce3e6
- a8e6abaa0ddc34b9db6bda17b502be7f802fb880941ce2bd0473fd9569113599
- aa94057d957736005bd6c70dba96b39b60121e0a4b35db03d5b9dfdbf5e58537
- adb9c2fe930fae579ce87059b4b9e15c22b6498c42df01db9760f75d983b93b2
- ae616003d85a12393783eaff9778aba20189e423c11c852e96c29efa6ecfce81
- b09d6ef6c8608adabf1c540407fef37e69da3d939daeaf7868e3802043ee7615
- b20ce00a6864225f05de6407fac80ddb83cd0aec00ada438c1e354cdd0d7d5df
- b5f5b25b3e93394d530df6ecb4f3f66bffb72af73fbe61859dc15e73ee43e9c0
- b74bd5660baf67038353136978ed16dbc7d105c60c121cf64c61d8f3d31de32c
- b83aa5ad1de3419436600f01d193b488f623ca58ac096eb8f365d5df3cd980d4
- ba63ed825b0dd3909865d54f075175b85073467c993a030ed66069f7532b3151
- bc2f07066c624663b0a6f71cb965009d4d9b480213de51809cdc454ca55f1a91
- bd6ad41421deec0ce042334519b47c212592044cbdea2fdfde2d385f6d1b530d
- bd9146a2dfb87cbb8b301917a21dbaa8a7de344f7dfdf3899b74fe86eaf43350
- be6bea22e909bd772d21647ffee6d15e208e386e8c3c95fd22816c6b94196ae8
- bead710db3df029cb39f9ad8a42620d05c44ee45a6fc80bdc9f75684954cd55f
- c2539cb0495fc09f1ba8b29c6eec17af61f502d4406cc214a0ee65211441efba
- c25c1455dcab2f17fd6a25f8af2f09ca31c8d3773de1cb2a55acd7aeaa6963c8
- c3c0c95173c488261bef90729dad997babf68fcf75c5013a543bef4542e15b1b
- c55cb6b42cfabf0edf1499d383817164d1b034895e597068e019c19d787ea313
- c946d8dd80e1efa8ae14bb84060e21b5d19055c68bafbc5512a3af0382cccd7b
- c987f8433c663c9e8600a7016cdf63cd14590a019118c52238c24c39c9ec02ad
- cb09c377721de670a698db9d56716be19946225ed7eb3dfccef283be28d7780d
- cb58396d40e69d5c831f46aed93231ed0b7d41fee95f8da7c594c9dbd06ee111
- cd019e717779e2d2b1f4c27f75e940b5f98d4ebb48de604a6cf2ab911220ae50
- cf3b528361557500dde295ae01ab84d1b37496d7240210fd6b114dfd80483360
- d066dfffb3d0dee40cf81cbcfb6209d09a57de33a079bc644456cd180c5f170b6
- d26dae0d8e5c23ec35e8b9cf126cded45b8096fc07560ad1c06585357921eed
- d344b132097288f944aa9bb57835f135f60615438bc94e2a232dee308a21805a
- d3ee530abe41705a819ee9220aebb3ba01531e16df7cded050ba2cf051940e46
- d5687b5c5cec11c851e84a1d40af3ef52607575487a70224f63458c24481076c
- d5e3122a263d3f66dcfa7c2fed25c2b8a3be725b2c934fa9d9ef4c5aefbc6cb9
- d8df60524deb6df4f9ddd802037a248f9fbdd532151bb00e647b233e845b1617
- da3c1a7b63a6a7cce0c9ef01cf95fd4a53ba913bab88a085c6b4b8e4ed40d916
- dc612882987fab581155466810f87fd8f0f2da5c61ad8fc618cef903c9650fcd
- e0cb31bb04ceb2f5d64debdc60a312981ee4b57444d1cb8d0dcea271e8fd315c
- e38372681d9fd42504d33a7956865bdc6e0fac15dacefb857c3bc279f7d6ad1f
- e3a7fa8636d040c9c3a8c928137d24daa15fc6982c002c5dd8f1c552f11cbcad
- e64dd4ad00b8615bca6917c7326fdefe791446929d866d6c2c7c13674f0d935b

- e6ecb146f469d243945ad8a5451ba1129c5b190f7d50c64580dbad4b8246f88e
- e7ac03729bc141cfac40bd7dadba7abd3c714fb1715034b44e28cd5561336a3a
- e7bbdb275773f43c8e0610ad75cfe48739e0a2414c948de66ce042016eae0b2e
- e9af4018616e4275c6b6af5531bb988431c1454d8567cc4f6c7d2b4dc63440aa
- eb8eefea77fb258bde014c3dfd9dc92c9b69598ecdbd74750d0ca609afc8808c
- ebb16c9536e6387e7f6988448a3142d17ab695b2894624f33bd591ceb3e46633
- ec7091913a753578f7485f6c22b73457c47862443fd4fe62709471aea1c8614b
- ece1610907c50cdeaf158d0ec13aa8b4aa31a8b831db8d3791da1e35296aa527
- f1d6e8b07ac486469e09c876c3e267db2b2d651299c87557cbf4eafb861cf79c
- f40ca2de6f1010a3cde815635cf5e1898095d97d27ca7b03abc9a76ab8a678ec
- f5e444469407a3e894d368b79878a149696015ed2f666dddb49bd484f144d104
- f6a180cc3b31693739089a9966dd1feb107bb49216f1e3ed11baab8e4f6b5226
- f6aab09e1c52925fe599246dfdb4c1d06bea5c380c4c3e9c33661c869d41a23a
- f79e4adc2cd11f9e44023cbdb827777a0c44af44bb19b494bef2d2d8e6e3be02
- f7f4d18dbc0b822b89ba14ffea24114f92b593be0f287f300bb269b310883039
- f94eb96f380a47bac95cb453e690ca78ae9ae1d078f8e2a433635a63bb73785b
- f9f2ac8167051aa7d2c38044cbbbd5542dd4901593aa0dcfe1a6ea9a92850da7
- f9f2b38e11402b56fe05127bf0e688d74bb6e55834b93b7a0f6c61174670177a
- fa116cf9410f1613003ca423ad6ca92657a61b8e9eda1b05caf4f30ca650aee5
- fd689fcdcef0f1198b9c778b4d93adfbf6e80118733c94e61a450aeb701750b4

C.2 The Hashes of Dataset DP

- 027b1042621f86394fd7da27c5310e4906f41b96f6e5474875e63d39b32a9c11
- 04c9240d425bec07742dd99d6f75e2205383ef804f2410c8274ff2e74be74ad4
- 06b883f27497b09bfd58240e16b4ad5310669df179ed9034e368b2afe26b7a24
- 095cee8f9f9f533b315843039a901d3613a31e6a0ae3322f52ca8711f8e3507b
- 0a58f3a7f988d57407537f77c5858afab5f042a78baae2b6d0268536e62bcc54
- 0a812976b9412ed28aee3ac3de57873fafe1ddfa0e6b9026078017b810d1b24e
- 0ce3bfa972ced61884ae7c1d77c7d4c45e17c7d767e669610cf2ef72b636b464
- 0e7d3772de05d030ca4c0083e2f48be06cfab01db0ab9091916ddd275765e9ba
- 10b94d3088a21b367c085e5f6493f022b47f279352e657719f7b8a5957964a1d
- 116b6154d04260ca235db78f2abbc647cc80b92a9838360eaae4f3b8eb50d5c8
- 12dc5c7b9c08f0654f31c274ba84c39af5ab8514b762a07b7b48439323f85bcd
- 13e40ee7c6874e2f1ed58bc09738a5525f86361f1a85387b2110f114d8f3272a
- 180e5227aae20fa2d6ae421835dc7d92f9393681c3006213dc2f6e3fbd07e3de
- 1ba4f8d569dafdf2c0152d706fc9cc3d6eb646e8ea639c410c8f95e07bc2551e
- 1bc9ab02d06ee26a82b5bd910cf63c07b52dc83c4ab9840f83c1e8be384b9254
- 219423a32336987838bea44a471fe02700e2e74ba4c98ebb41512b7bc15e0c32
- 263175e8161c374528de87a3f70b9312aed4d16835d75d421523a2ad5c392d33
- 2635a89660d6c99fa852258704e00f097f24c10343bb523f1e212dd09835459a

- 276b17244408e7e698e837a0a105c7c3857acf37e2e837d4b10e6904fd9dc3
- 28ce6aa6b39743680b5fe34bcd9822db3d30716cc3faf09bf37d0bc05a73d9e
- 29fc21a0c2cb5bc631a730f9fb1379ca9847746d1dc1f30c99598f7c96874e86
- 2badeadd8bdd305d1a9548ec599032a9d8bd25ed41357eba268aa13cb9ffd8e
- 2cb5ac355310c95b3792acd173bcaa0081219646d2951cc0a3e056d5152e4a5d
- 2db1ec48d199b36e59726b60a4e1f95a3bb5402e4f8dea6ccb5dff973d77bcc
- 2db8a9c401911c7317e8a89c35d979d0e8e9ba718ae13a0a0cfedd957654ec10
- 2df5bbe0e055e2af7d32e3b71ea80b70f844a917229a6b7f9668eca31c3d813e
- 2e8d265191a86af4195ff0cdc24113d74369a05128a72b5212cbac6d7f94306c
- 2efefb0778b265c7b7b87262eb615014d1dbb90a4a64255651c67b814feee057
- 2f73ad3dc5bc16ba73876c8d9ba6c744cf9ea695cc644106165c72a5eb4fd79d
- 35f911365d14ff533acce7367c2ab74167a9beb7b4e8fd487f25b9db4d68f627
- 360afa487a76edcf1aad4d6d7068740e3f8c6c1a8f04bf2cf5351db7de344570
- 37db8b987286897ed3a82ed93370279a01df428efbf0fb4dcfe4452480f537a3
- 399e19d2dc5e96531666a8cc3071115cf9b19ba1d1676294ee77bd2c75d25add
- 39c05b15e9834ac93f206bc114d0a00c357c888db567ba8f5345da0529cbcd41
- 402557e597c5f93cb35055c43335be5e7ab9de9cb088f3cd003e204ada2fbfe2
- 41009a7ee00d2c640e9f8681f65352b85eebc43f5536ea078ac91372a60f5ee7
- 416b97dedceabfd94cfb32315f1997611d6530a1bec939959a1b0c504a63b224
- 422ba6dae6752430a2e52e1efb327f277e912ce551f9f1408ee6ab13ebf3717a
- 44a700a18b4cf050bfde1f9218b822bb37c770d16431052bf827f2544cd51ec0
- 49bc860fb8856436e1d540754732843f1a534901ecdd031870702bacab58ae54
- 4c2efe2f1253b94f16a1cab032f36c7883e4f6c8d9fc17d0ee553b5afb16330c
- 4e31304e1ea66c267b5882f9335a2384eea18a6617a49308846ce624b68e7489
- 4f55446d65578f9c0ac2694ab2f07af60694a8d96e0acb484aac192d58e819b6
- 4fa0e0cfe9406f6644613f91afda3e48418f147e0145712a3ac334492edba5af
- 5262cb9791df50fafcb2fbd5f93226050b51efe400c2924eecba97b7ce437481
- 551af522d2adbc24c3821a3408d231045da0d4dc55ff559b0c8049d36d10a16d
- 560cb7ab7ebc67979def9ef411a6a1f2a90a0d53c2feb22c020f5623ef40051d
- 567a50a08529d19f146c1c3eef09d22679a358d15fdf6f508897b35b95e75e6c
- 5a9d84792a06b3d2037f567e0f57781722a950d485854cf5e4042cbdd51d82af
- 618a75808b11fba4d1501587f2df23c6bf4094a474497a1f15fb85bbdc6cd593
- 6392e0701a77ea25354b1f40f5b867a35c0142abde785a66b83c9c8d2c14c0c3
- 643a99f14cfe5b4a9ca9dd5a30480ee9c115e2dd82478a725f20988241958a67
- 655ca4fa15b26643deae1c02b7a31d2dc6971e317f757c4c222057255dcbb580
- 6619a4ff7f0478f8c15fc0391651a1694afe876d25ebd07e3da08167e4f0b3d3
- 69061deb711b6c29e51e37808c49699741c0f923b15391f073239cdb1a295e27
- 6ac981d137569bf3ea8d8d929a2e8c63acfbcb7bc87de8521b50e2874990d5f2d
- 6c095b01ee712bbca41dc10d9bcc7875db2a87b1fa9a71f60b39d46f2b87983b
- 6c7e60bccb286283ca1b839aa0be2c3b106dc70f4290dd99357ede189bd0201b

- 6fabbbf51a4171a195b9c7cea98902d7b9fc3993aaf44ab6967ba48543b1fd893
- 6fd002fcdc1a8447f03c227abd3e6551f9179ed79e39591069bca4b9fc9d6a8
- 70480daaf97bfc10fd793ffea9e90e1fcb84861415d14a3766a238a29cf30f7
- 70ff05fdeb51559c17696eb1c8577dd0aed7eaa6b6922c711aa0b6721db246d9
- 73794263b657632805c8c3907e2f20a9743d8c9b83aa3e21629eccc5de02b1ca
- 7477ccdc1980440879b46ecf2be5112b5ebb04a0baa740cdaa63db5ce822143d
- 7612c9240a766c427ee63cdd81c434bf646070792ead8748d3dcb2d1d326758d
- 7a3b78feba1670850602b7c33cb0968b4d89db609d98c81744b43cae23d563f5
- 7abf424fd57e49756307cc07e05627470a0d1f000a3c8fcc422ea4391981f6a2
- 7bc1ca150bd934d6a8dde2f8fe6c88eed6b6d56cf0c2b941aafc40735f0a0eb6
- 80e1e8c7b7e69a46a97ca4e6f591b15e09ae288b4e0bb5ee457b26c61062da92
- 8119f075b901142e437224b2f4fc059d36d1080b31b3f92a68400c10c1fa3d56
- 85bca7ec25d2b226be5e8e20d51fc7718366f70291a398b6aa617d734a3e7ba1
- 8a1da77a8b848f852c32dd98a548d24f40c8654f39bd30ac5c72ceb1e4f2d286
- 8a41179a750b11d09c4e4c251eb8f3927d2bef4d40e7e15594dc359783bdf04a
- 8a5c431904b4f7fbb565c9d1ca0faf03e1c847f3cba43eab5f9da47d7a8c897a
- 8b7427620d6537aa905727af48f7dec1e003a8b7c74d417f0a5ded7926a7d590
- 8cc3ede145613b926268828965830ad7fbcf0b6db2b8772b4d485a55b88dd308
- 8ccfd254277b451df5011669be99302761f224fe282a05c450e5320b3c77f2d8
- 8ef13ccf86c1ac1c2fef370a85b7c576afec11cf056c7d4ec288c126368f115c
- 91dada758659b410889a8a31c2fa04bed18e0eb6ed20c253b436b39f2bef0dc8
- 98263df30ea803b15a3b4b3ecdb2761c8d6148c833209513148912b974e0fd5a
- 98f028dfd1ef15f10c1184823ae7199e329aa3c811d511d183ee83e68af3c980
- 9a0bab44fd1d1364621c4fa76a43cdc4f7d3ca5a1961dc202e5abe2147785f98
- 9ea0a986c1bf49837b2413735860e66a79419f12711780b35e0b182ca3a1c79c
- a1b80abf76fe66c4a98d8f5e091ef5cab00ebbddf2aac9e1351d1c040568a0e7
- a4ddd6bf7d4095d5f3f8053db5dcdf7637badc02ae55688a29f541154b6d6ee6
- a80522d3a11f95ff57c74d45f99c48aa7aeae2f0c8296a52541ca5e87f0ff45e
- a9809c4464e571112467e5989d10b59b0af049e816dc3cb16f5b7128d5a3cda2
- ab1b410897b2d2fc56e7036fc5bc826ce142bffb5fbe18d94c1eac81445ec39e
- ad410f7e25082e139b433814c370750ae74cf43727486cb2e0b35ad88b2c0910
- ad4b4bccc23b312f62461e80250c82afd1fe3a0910fcfc94f197d2803bd1c30b
- b1aef85f454e00f2c6b982bdd6ce81d23d28701bfad7767252f9c64c1bdd6051
- b2417de25ad9e6bed08229561eb96d4f2e83ab63b4407c7601a0113ed193fe84
- b2f3323418d20b0b91419e58b6ca2b57423286a7046f6729cafbe39178f65124
- b41747714910cee5eb306f61dfa61dd5c3c72450a60fc36280b8d7fd0643b54b
- b719da8fb6f8911f02c1ebc83e3f2bf9da425fd634505918f0fe93d8013d1817
- b8c4976c8edca381f729f50a33b33d75930e5f8fb790d70f11853adb91448176
- bd11592557d2dba4e2cc5cdfdbc61cba64735ae01050db58557e2281389512a0
- be4fb3149fde2a18c68a3bb85084fff9212c5a717f89e4ed300929a4e2eb301d

- c1e9dfa1f1b3037da9b72354edf25250c12084234bccfbb6d970b1c196cddda1
- c50a48ef605b1f57f37afb883d643d69233cf506065d2bf806dae639cac8c264
- c52f34dd30dfd7c232aa835733ecaecf6905fa7f401f90372fdc4db4e02b6ed3
- ca5094b2dbd7a0cc4531034955d4563c0504e1b4ea262ce6b6ff023fbfc06f1c
- cb0966893437f5c00180202c2061b6532a8d8487dc2c69bc8b297ea08d2ed180
- cb24bcb0abb48a8e21d48cadec15be0bdf390b19c00285d4cff51ecbcf45f51a
- cd4c504294725b66262756314f36f0b1dd5db0a7e018506ec2cf4b38d74e1448
- d3931ee10daf52359a7591418690f97d4dd2c053624b231358e433f9e58769ca
- d4a15ab2af2be3d1b5697ffc27d5532b1dbc0b62c9466b6a1911386faa8f1d9c
- d579b32bedbe846c9d3c89aeb8d0c33ac22d6c0d9a1d3345b0203eb351dc7f95
- d6afb2a2e7f2afe6ca150c1fade0ea87d9b18a8e77edd7784986df55a93db985
- d7dc425c79aaf63ad582786162609edd02ace53588133b7e41f0d4a79193f2cb
- d88bd6947eef00bd3baadc55ff1c55b3cdcff5ba8fd145d5b5bf8894c42a7fd3
- d9cfd9e64cdd0a4beba9da2b1cfd7b5af9480bc19d6fdf95ec5b1f07fceb1d
- db1483528df7803a766f5a536c2680ac9ca0f6c8566753adecb8fd0612682d98
- dd003aa02f3c7ef6179e6b79d868d62afcf273ad3c79f99ccc779c12f46a17b2
- dd29a6b5c62d8726a3073b6f7d20e6f34d00616de61fc55d04bda9e7824cd598
- de18a47320a1eb08efd96e7bcee8ae0b3cd19683bc602063b854cf96a51536f5
- de4ff8901766e8fc89e8443f8732394618bf925ce29b6a8aafed60f496e7f0e
- e153e214ec22754fd6bbd4d4b62b87651216badda2d5c1124387aede2e1d66bb
- e1f52ea30d25289f7a4a5c9d15be97c8a4dfe10eb68ac9d031edcc7275c23dbc
- e547e8a8bc27d65dca92bc861be82e1c94b9c9aca8a2b75381e9b16e4ad89600
- e575c39549529d79d3346a5bb09cf7b484083a83c56db65c5db686a41da9a2bc
- e917d277ce6d27e9740fede690f7bd810e99c0757ae4226cb30f8227c6b30b43
- edf508eb8566e9c8b519661bc54319e9da40b1a5cbe8da0c2b5a3c25fca73a39
- eff3444317ceca3b4642ee4ad3ed947f7bb17e35976465fad686ddd52cfe8cc5
- f10e3c8ae94b4ee00f6a09e72a9051d682366dae58f3bb7a7aab9c9b99b7714c
- f5caa7bc9098f04fcc8c14cb65943b8c2b711901f4732f6496617f13b8d2ef7a
- f79e4adc2cd11f9e44023cbdb82777a0c44af44bb19b494bef2d2d8e6e3be02
- f7e9a1a4fc4766abd799b517ad70cd5fa234c8acc10d96ca51ecf9cf227b94e8
- f9f2ac8167051aa7d2c38044cbdbd5542dd4901593aa0dcfe1a6ea9a92850da7
- f9f2b38e11402b56fe05127bf0e688d74bb6e55834b93b7a0f6c61174670177a
- fa116cf9410f1613003ca423ad6ca92657a61b8e9eda1b05caf4f30ca650aee5
- fb69587a8747dd2e7c521180f2ac2bd156a2e9ad83b7e1b47d65c55e2d8d395a
- fcccc611730474775ff1cfd4c60481deef586f01191348b07d7a143d174a07b0
- ff83d9796124bda7e50f9957858f3db688948127c2e1a7bcb6be79b25baec2ba

C.3 The Hashes of Dataset SP

- 03641e5632673615f23b2a8325d7355c4499a40f47b6ae094606a73c56e24ad0
- 051f9ff45c531ad265489f563e6babca55f4a3f94604ff56e37140743f30badc
- 07529fae9e74be81fd302d022603d9f0796b4b9120b0d6131f75d41b979bbca5
- 0832ec4e7a6e59fe03fe7d7614eadd67ceea3f330b309cadb4aacaf05d46ba61
- 0a013787f9c1731213059f2d8e1a7514f610783aaaaea8fa5736063ab7793c0d7
- 0dcf284acee023eea3564ab3f858289ca8701ee400cdc36647f03e7b681de915
- 101f1171f1cf0d7edd76e15608bbabf51b1a6e2f5f96d5188264eb8325594818
- 1039f5492d975d1f215255397ebc2419fb136623682ee004e39970fe9b84dce3
- 1233cca912fb61873c7388f299a4a1b78054e681941beb31f0a48f8c6d7a182b
- 129b10324274afaf7e99e4973fdad83fe098d02726e1ad1c3334fd99eef7f1eb
- 131314a6f6d1d263c75b9909586b3e1bd837036329ace5e69241749e861ac01d
- 13fcf8519a6108ce7e826feffe2c46447cfb2f093fb961e6a2fdaff2a41499f4
- 1952fa94b582e9af9dca596b5e51c585a78b8b1610639e3b878bbfa365e8e908
- 1a7239c006a3adf893bdb5c2300b2964ed8bb454e1b622853e4460707dc63c16
- 1b76fdbd4cd92c7349bc99291137637614f4fb9598ae29df0a39a422611b86f8
- 24a9bfbff81615a42e42755711c8d04f359f3bf815fb338022edca860ff1908a
- 2db8a9c401911c7317e8a89c35d979d0e8e9ba718ae13a0a0cfedd957654ec10
- 30196c83a1f857d36fde160d55bd4e5b5d50fbb082bd846db295cbe0f9d35cfb
- 309217d8088871e09a7a03ee68ee46f60583a73945006f95021ec85fc1ec959e
- 33e6c3f5a66512c136e53ede2095fc240973fa58a9d8a7b69f23db01c53f2f59
- 3577845d71ae995762d4a8f43b21ada49d809f95c127b770aff00ae0b64264a3
- 384be453cd26f60cd2358a09ccd64e4e7a27c1f8c6da5df557f8cc4913686112
- 388f5bc2f088769b361dfe8a45f0d5237c4580b287612422a03babe6994339ff
- 38ce3d3bb90e6ec890d3887b9e460c6f0ea459ae5fb87f583469f93953355e9d
- 3b2b1a923caaa15985d51d1cc560001b598e90d9c06982df93e4cc6201917449
- 3bedb4bdb17718fda1edd1a8fa4289dc61fdda598474b5648414e4565e88ecd5
- 3d3da3e3c062b5b73de6dac3cef8a90c4c7eed5622dafdd9a47ea35af9056297
- 3e49de7fbfb33c3ae715f948cf1f65d1eae5f4527c2b423be886fe7194fc5c34
- 3ea6b2b51050fe7c07e2cf9fa232de6a602aa5eff66a2e997b25785f7cf50daa
- 42fdf3714f5b2598e348ce4e6f57937f831fe6d3b3c6d975854f66407567dac5
- 45985ca9eb04546745cec575d95b1684adc71130d1e0a8db2e36ad6d8df8b2a8
- 4c01ffcc90e6271374b34b252fefb5d6fffd29f6ad645a879a159f78e095979
- 4d3a0ba910024c6ca1ca9e915eb43fff7f9610406105750383f716069e7dfb91
- 4e4bdaa67c368ce41f8d4ba8fecedb9a9d6177a9ed9e0ef45312c9c33ee4394e
- 4f02a9fcd2deb3936ede8ff009bd08662bdb1f365c0f4a78b3757a98c2f40400
- 4f4fa26bc26fd90c64dd3b347a92817b67b64506c025248330aa69b00b97051f
- 50414f60d7e24d25f9ebb68f99d67a46e8b12458474ac503b6e0d0562075a985
- 52cb02da0462fdd08d537b2c949e2e252f7a7a88354d596e9f5c9f1498d1c68f
- 55504677f82981962d85495231695d3a92aa0b31ec35a957bd9cbbef618658e3

- 5663b2d4a4aec55d5d6fb507e3fdbc92ffc978d411de68b084c37f86af6d2e19
- 58da2fe190b50f90484753f92f9f61a13c9217e593e55b6dafa32b12dc049
- 5b2d2de9a95add2b71f3a9aa6c02fa56555b7d58270fd073384187f52b76a603
- 5b50e26a01b320f05d66727e9d220d5858cdac203ff62e4b9ced1cafc2683637
- 5d491ea5705e90c817cf0f5211c9edbcd5291fe8bd4cc69cdb58e8d0e6b6d1fe
- 5dcdf2e8f1b9348bfd3330a31a70a4b5fc03dd86e45553dca9d85f74f9d8ec6c
- 5de8c04a77e37dc1860da490453085506f8aa378fbc7d811128694d8581b89ba
- 5eced7367ed63354b4ed5c556e2363514293f614c2c2eb187273381b2ef5f0f9
- 5faf76b8b06c727a08b34e456ddb792797fab734cbd878136d85a1f767d8875
- 626eb96a340c865ea8ed721c94fac4504db147b0992e0190438b7cc144c05614
- 65fa52f632e4e83ff83120c7df6b90291025a76d5daeb183e814ec0b3bd2bd4e
- 690b483751b890d487bb63712e5e79fca3903a5623f22416db29a0193dc10527
- 6ac06dfa543dca43327d55a61d0aaed25f3c90cce791e0555e3e306d47107859
- 6bflaea8203ec5dffffef99e222fc39cc7072c71472ccc28611425818794f091c
- 6de091db9dfceab8b867b3b52953d270b69eea7529d400d144d6339bcddac01c
- 6eeffe540693418a107db3e7d2d9b72a54b2354aa6886b571272aa41f8cc8e0c
- 71c813f0e0a96b5b85c5d798dee316794fc4715703656732e3d42518c8f141ba
- 73aae05fab96290cabbe4b0ec561d2f6d79da71834509c4b1f4b9ae714159b42
- 741926cbac03dce1af5156843de89bc89408d8a7588451716232b2a203fdbca9
- 7522bc3e366c19ab63381bacd0f03eb09980ecb915ada08ae76d8c3e538600de
- 7581d381c073d2b67bf2b21f5878855183f9fddf935557021ee6d813b7dda802
- 7dd063acdfb00509b3b06718b39ae53e2ff2fc080094145ce138abb1f2253de4
- 7f26bcad404867f92ee0f3de9257758132b2ea06884f436e7900e820ddd6646a
- 7fb1a6f23d30495ad7367d38aa763f36701cbc476c868fc9d37a946e9bc26d53
- 815854f24334183ac78eca4edd21773858ef199df578b931c2ef619f5e8f8887
- 825858c467b49923adac73ee23fd972ac5fb709690ec605e153315398b991dc1
- 82d48ea61901b6412fb7ae648ec752fb29d61ef520f33e37ae7a058fd99a95e5
- 8585342d297b4726900e8818817b14042e1a3da5a1497380572a64dcf6d4819c
- 86056f462d5783604b7f050047db210ecf698e72f3664b27d58265663ff5b324
- 890485744eabef256d6a2353c7544ec4f2d7bce27c9969998b7557afa56a084c
- 8cde65e4091da0b5e51df00ac2fd604c89e3256f372a8757c94910228f90ce6c
- 8f0674cb85f28b2619a6e0ddc74ce71e92ce4c3162056ef65ff2777104d20109
- 8f0a426fc1b7885a2785c5442d12b9387e84a8ee75daae0234d78b513c7b5c8c
- 92d24128a45f33bdca5f28eb0319668cb97fb2f8a7e7b72d70a3aa4c897a4975
- 9637b2dcd5f9d5fdc0f1c1104f73f3dbdcfd803cac47196cc94c768c21fa2ae4
- 98b6371c901856a12d1c6295c4026f93747d133f4e4969989d8de83fe9d6418f
- 98f400ccedd112792e562728a4d66c7c18712684f4729014dbf6fdee2931eeb9
- 9d9e8ace43981a3d69f96f6471fd39a30191d849cf6f55151834bcadba1e9321
- 9f012d7e3ae8f62370278e372691eb73b878fe2280b6083e1be637b278021855
- a42030e9b290413b58d8507a528c1e8fb49f9d602ddeae9c03785a1f1e8497f

- a4d70f2cf36a00e1985a1275020078c8c4fb472c478aea708cd271ee4163009a
- a52762177877479859e4f88a13f605ad1e69d759019cf49dcf026781375b74a7
- a908f128b77e8488fd2209fc162779b05ec9b633493751a0f30b2e5664c7183b
- a98099541168c7f36b107e24e9c80c9125fefb787ae720799b03bb4425aba1a9
- ab3ee0f2914deb0098a2cbf756cf0fa06db71427a8c9b18a72942ef41014543e
- accf700ca998053f160997f5414392372698f991ae743aee1165a78fe9a370df
- aceca16c33ae8a73b1fd7699a8317d70d164df9744cb7e494834b9c1e457a768
- ad5ce793bea64ed9a416236d1e4b99b021a771e5d57a46b97cf1e288e5617132
- b083affa61742de62cdf8301f777d074d3e23c9e42c35375672d9560c77f8bc6
- b0deda9f8354364d78134de41f845a194f29dc6e24f6e68c6d959e2e518f3791
- b275c8978d18832bd3da9975d0f43cbc90e09a99718f4efaf1be7b43db46cf95
- b3d3fe54f71d41414232c342c37f539651ae3ee49ec2d47789cd2c71c6271b48
- b690394540cab9b7f8cc6c98fd95b4522b84d1a5203b19c4974b58829889da4c
- b9c996b06e0db273a4edede3fd6fda2b40b2e0201eba3e8ac581d802fc610a4a
- bc12d7052e6cfce8f16625ca8b88803cd4e58356eb32fe62667336d4dee708a3
- bc452cc1128ccf7fa9f76d83cda79132740414973600fed14509749fe946816e
- bcd43ed6a838a35dba5298594ffd6b86d4dc804c7c524e541e5cd4c92fb8fe72
- bd35b10e076fe95e122a3dd7fe93df7621cac7e3c32261adcf45a81c10f6f692
- bdef2ddcd8d4d66a42c9cbafd5cf7d86c4c0e3ed8c45cc734742c5da2fb573f7
- bed0bec3d123e7611dc3d722813eeb197a2b8048396cef4414f29f24af3a29c4
- bfbcc50ce93a2610ec5be137d07ee1b47b2511d3802771541c8842e2f3a908c23
- c1dbf481b2c3ba596b3542c7dc4e368f322d5c9950a78197a4ddbbaacbd07064
- c4054c514f0c58bcff456114768723b257013c49eba9ca61e395484e81e01d19
- c7212d249b5eb7e2cea948a173ce96e1d2b8c44dcc2bb1d101dce64bb3f5becc
- ca42a173acd560863c6ef759e59a7e2f6c4761c0c5b8b2f1e8df3376c3234824
- cca3eee2650d20cf1bf50b76e7f97a3b0e26caff3af8546462c92f2e73d730f9
- cee21b4199974ba8b4c77bb9dff62bea9b87bfde9df4c89072c3d12472f65861
- d0f059ba21f06021579835a55220d1e822d1233f95879ea6f7cb9d301408c821
- d426869f3dc8c7ffa65d1cf6e4fff8470ac5c0b39a03daff4d6caa0ac806e7c9
- d5debe5d88e76a409b9bc3f69a02a7497d333934d66f6aaa30eb22e45b81a9ab
- d7a159a8a84ab15a9a7572d7cab56b1a9de29aa0cb20aa3d7fc010f4c547ee1a
- d8fcdcaad652c19f4f4676cd2f89ae834dbc19e2759a206044b18601875f2726
- da48960557dfa3aa41b09be150ce662daeb38133d981baea15bbac02fdd9aeeb
- db36fbc490ffb8662d13388cf4a1d5daba1527ac0440fc2eff99ddedc51f4a82
- db921a575fa7fd4b0c1b405a54f77d10c73eb1cb1384a27d584d7323e72938b6
- dcbc9d204549f768a24ec821f2b709c86caf0f5821514d9004b62443ec3e6933
- dd5261df621077ed13be8741f748f61c5ed09bd04ca48526492fc0b559832184
- de33dfce8143f9f929abda910632f7536ffa809603ec027a4193d5e57880b292
- dea53e331d3b9f21354147f60902f6e132f06183ed2f4a28e67816f9cb140a90
- e1805fa3c90f07f9721ad4bf4f42caf89cc5c3d45e0240c98cd34426caa4e670

- e2e6ed82703de21eb4c5885730ba3db42f3ddda8b94beb2ee0c3af61bc435747
- e458031e9cee02dc4b7a9404d6dd3fcce5169ab13ca3e915357d45816af4e9f2
- e50407b62502bfc2fe94c97e0d1af3871269596b8de3384df4dbb92f90de17c6
- e5b68ab68b12c3eaff612ada09eb2d4c403f923cdec8a5c8fe253c6773208baf
- e61e56b8f2666b9e605127b4fcc7dc23871c1ae25aa0a4ea23b48c9de35d5f55
- ea43d8973fdfa6eb77e1f6b6a5276c5e06c614071d26f68f19fbc25fe09aad4
- eb1079bdd96bc9cc19c38b76342113a09666aad47518ffa7536eebffaadb4a
- ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa
- ef47aaf4e964e1e1b7787c480e60a744550de847618510d2bf54bbc5bda57470
- efce8c3f7dd1ee2b6c40fd84909e5d423bd75a908b546fdb7ef73480eea2871
- f0f7a1997a1ad57ce62bd32ace27304a6b925af8b63513c8007181e8bb5da919
- f4d68b4ff8b0ffd40aba886da64c5a82cde7c489d7973cc72f5f7800030ad883
- f9d94c5de86aa170384f1e2e71d95ec373536899cb7985633d3ecfdb67af0f72
- fc085d9be18f3d8d7ca68fbe1d9e29abbe53e7582453f61a9cd65da06961f751

C.4 The Hashes of Dataset TP

- 027cc450ef5f8c5f653329641ec1fed91f694e0d229928963b30f6b0d7d3a745
- 0ce3bfa972ced61884ae7c1d77c7d4c45e17c7d767e669610cf2ef72b636b464
- 1233cca912fb61873c7388f299a4a1b78054e681941beb31f0a48f8c6d7a182b
- 1a7239c006a3adf893bdb5c2300b2964ed8bb454e1b622853e4460707dc63c16
- 2c480399bff7d05736caa1858fd43d9223df3fd531ae574dc3c9eb06cc3579ef
- 2eafc64769c500d635b7225c9b1411db8f50db8618e4d5807e1640b641a2f5ee
- 4bc8280a99d07165055fabed11049d8da275f27f5d8cfff4ed10a68be2d0cb84
- 4e31304e1ea66c267b5882f9335a2384eea18a6617a49308846ce624b68e7489
- 4f9b6a88245f782d81e9eec9315b9444c83d68941f9fc23641e3909c8da9db9d
- 53483cac73b9aeb9985d4408226eb1ef031b8b882df8d8a3872308d33d3be705
- 5ef73d904cf5dcbec5919fba0b640168d6feb8f7021507568297e3da1a7e47a5
- 65fa52f632e4e83ff83120c7df6b90291025a76d5daeb183e814ec0b3bd2bd4e
- 6eeffe540693418a107db3e7d2d9b72a54b2354aa6886b571272aa41f8cc8e0c
- 7a3b78feba1670850602b7c33cb0968b4d89db609d98c81744b43cae23d563f5
- 7abf424fd57e49756307cc07e05627470a0d1f000a3c8fcc422ea4391981f6a2
- 7c4101caf833aa9025fec4f04a637c049c929459ad3e4023ba27ac72bde7638d
- 81de431987304676134138705fc1c21188ad7f27edf6b77a6551aa693194485e
- 82670519b8d63d36967c611bc94659e5bff867837129ac93bcffe7589af46384
- 86056f462d5783604b7f050047db210ecf698e72f3664b27d58265663ff5b324
- 8ef13ccf86c1ac1c2fef370a85b7c576afec11cf056c7d4ec288c126368f115c
- 910a016a7b6e0a76bc7ddf12f9135090e0b23d00c382d70084b46bea4bbbcae7
- aecb468db5cebcfa25deadeb3b12fbc48b05a485b44deb500b4002521bc3e685
- b2417de25ad9e6bed08229561eb96d4f2e83ab63b4407c7601a0113ed193fe84
- b9c996b06e0db273a4edede3fd6fda2b40b2e0201eba3e8ac581d802fc610a4a
- d93f22d46090bfc19ef51963a781eeb864390c66d9347e86e03bba25a1fc29c5
- d9cfd9e64cdd0a4beba9da2b1cfd7b5af9480bc19d6fdf95ec5b1f07fceb1d
- dad4c4aea24f2bd3e2f4b93bf782ebef70e8fdf930aff25a3e1b85a717314aa0
- e1f52ea30d25289f7a4a5c9d15be97c8a4dfe10eb68ac9d031edcc7275c23dbc

C.5 The Hashes of the Binlex Dataset

These samples are used in the experiment in [Appendix A](#).

- 4729a6b153dfa2ad96bbae68aa2a7cc0b21174013ace970386afb5a087951481:
9ff8e68343cc29c1036650fc153e69f7.exe_ from Capa Test files (discussed in [subsection 9.1.1](#)).
- 75eb05679a0a988dddf8badfc6d5996cc7e372c73e1023dde59efbaab6ece655:
Lab06-03.exe from Practical Malware Analysis¹ [\[34\]](#).

¹<https://github.com/mikesiko/PracticalMalwareAnalysis-Labs/>