SEVENTH FRAMEWORK PROGRAMME
Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

WORLDWIDE OBSERVATORY OF
MALICIOUS BEHAVIORS AND ATTACK THREATS

D08 (D4.1) Specification language for code behavior

Contract No. FP7-ICT-216026-WOMBAT

Workpackage	WP4 - Data Enrichment and Characterization
Author	-
Version	1.0
Date of delivery	M12
Actual Date of Delivery	18/12/2008
Dissemination level	Public
Responsible	TUV
Data included from	EURECOM, POLIMI, FT

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n°216026.

SEVENTH FRAMEWORK PROGRAMME

Theme ICT-1-1.4 (Secure, dependable and trusted infrastructures)

The WOMBAT Consortium consists of:

France Telecom	Project coordinator	France
Institut Eurecom		France
Technical University Vienna		Austria
Politecnico di Milano		Italy
Vrije Universiteit Amsterdam		The Netherlands
Foundation for Research and Technology		Greece
Hispasec		Spain
Research and Academic Computer Network		Poland
Symantec Ltd.		Ireland
Institute for Infocomm Research		Singapore

Contact information:
Dr. Hervé Debar
Rue des Coutures, 42
14066 Caen
France

e-mail: herve.debar@orange-ftgroup.com
Web: http://www.wombat-project.eu
Phone: +33 23 175 92 61
Fax: +33 23 137 83 43

Contents

Abstract

This document provides a specification language to describe the behavior of code. Consistently with the requirements for an extensible, layered architecture for the behavioral analysis of malware, four different languages are defined, ranging from a complete, low-level description of the code's behavior to a high-level analysis report that is suitable for a human analyst. Furthermore, current approaches to behavioral malware analysis and detection within the WOMBAT project are discussed, most of which already take advantage (or can be extended to take advantage) of the provided specification language.

1 Introduction

Malicious software is one of the dominant problems in the field of computer security. Many forms of cyber-crime are conducted using computers that have previously been compromised by malware, such as sending spam e-mails, conducting (distributed) denial of service attacks, and hosting phishing sites. This is a long-standing problem, that has been tackled by numerous institutions, such as anti-virus companies, academic organizations, and even law-enforcement. However, no definitive solution has been found so far. On the contrary, the number of newly found malware samples increases dramatically every year, stressing the resources of malware fighting organizations beyond their capabilities.

Due to the enormous amount of malware samples that malware fighting organizations deal with on a daily basis, it has become infeasible to treat each malware sample independently. Signature-based anti-virus software heavily relies on finding signatures that are general enough to match whole families of related malware samples, both to reduce the work associated with signature generation to a manageable amount, and to allow the scanner software to run at a reasonable speed. Similar considerations also apply to network traffic signatures as are used in network-based intrusion detection systems.

With the ever increasing amount and diversity of malicious software, it becomes more and more important to obtain a global perspective on the problem. Instead of analyzing and fighting malware on a piece-by-piece basis, malware must be inspected for its prevalent features, categorized, and characterized by means that are abstract enough to eliminate irrelevant details while still maintaining enough precision to enable any required further processing.

One of the goals of the WOMBAT project is to provide data on online criminal activity enabling scientific research into cybercrime. To this end, large amounts of malware samples are already being collected by WOMBAT partners (as detailed in WOMBAT Deliverable "D06 (D3.1) Infrastructure Design"). Consider for instance the Virustotal [7] and Anubis [1] services, that receive large amounts of malware submissions on a daily basis, or the SGnet [52] distributed honeypot. As the WOMBAT project progresses and the new sensors, (described in Deliverable "D07 (D3.2) Design and prototypes of new sensors"), are deployed, further sources of malware samples will be available. As an example, client side honeypots such as Shelia [6] will provide an additional avenue for malware collection.

In addition to collecting information on malicious software, WOMBAT aims at facilitating sharing and enrichment of this information. WOMBAT Deliverable "D06 (D3.1) Infrastructure Design" describes the two main components of the technical infrastructure supporting such sharing, namely a centralized database, which is fed by WOMBAT's data sources, as well as the WOMBAT API (WAPI), a programming interface designed to enable direct queries on the data sources in a decentralized fashion. The specification languages described in this document provide a further tool for data sharing and interoperability between heterogeneous, independently developed research projects, both within and without the boundaries of the WOMBAT project.

It is important to understand the actions that each collected malware sample can perform. This is necessary to determine the type and severity of the threat posed by it. With the fast growing amount of samples comes a growing need for automated techniques to perform such an analysis. Tools like CWSandbox [10], Norman Sandbox [11], and Anubis [9, 14] have increased in popularity. They execute the malware sample in a controlled environment and monitor its actions. Based on the execution traces, they generate reports aimed to support an analyst in reaching a conclusion about the type and severity of the threat posed by a malware sample.

In addition to sharing malware samples, it is therefore useful to share a behavioral characterization of the malware, as it is delivered by such dynamic analysis platforms. In fact, in some cases, providing the malware binaries directly is not the most suitable way to share information. There are several reasons why it can be preferable to share previously gathered analysis results instead. First of all, the malicious programs themselves cannot be distributed publicly, since unqualified or mischievous individuals could cause harm by executing (and thus, releasing) them. There are no such restrictions on reports describing the malware behavior during analysis. Furthermore, even if a researcher is allowed access to the malware samples, he may lack the computational resources and infrastructure to perform behavioral analysis. Running dyanamic malware analysis on a large scale is computationally expensive, since observing malware behavior requires executing each malware sample over a certain time period. Additionally, since during a dynamic analysis session the malware is effectively executed, there is always a small risk involved that damage could be inflicted, even though all possible measures are taken to avoid this. Finally, since malware often reacts to its environment, the results of several dynamic analysis executions of the same sample on different analysis platforms might differ, which complicates later discussion and referencing of the data.

In order to avoid the mentioned problems and inefficiencies, it seems preferable to share not only malware samples, but also the dynamic analysis results obtained from them. Clearly, sharing information requires a common language between data producers and data consumers. In this case, the data producers are the dynamic malware analysis

tools that provide information on a malware sample's behavior, while the consumers are further manual or automatic analysis tools that take such information as input. However, these tools have been developed independently, and they often use specifically designed data formats. The inhomogeneity of the used set of tools and their associated data formats necessitates the establishment of clearly defined, common means of storing and sharing that information.

This document defines a set of specification languages with the purpose of efficiently sharing dynamic analysis results within WOMBAT. It suggests three layers of abstraction for languages meant to describe malware behavior. Each of these layers is suitable for a different set of applications. For each layer, concrete implementations are explained in detail. The *Raw Behavior Specification* language is proposed first. It describes the execution of a malicious program without abstraction, including the complete information obtained from the dynamic analysis. We also describe two intermediate behavior languages, namely the *Abstract Malicious Behavior Language* and the *Behavioral Profile* language. They abstract from many of the less noticeable details, while maintaining a level of precision that is appropriate for further analysis tasks, such as malware clustering or behavioral malware detection. Third, the *Behavioral Analysis Report* language is presented, that is meant to provide the results in a highly abstracted, human readable form. The discussed language definitions have been implemented and integrated into several WOMBAT components, enabling the circulation of valuable data on malware behavior within the WOMBAT project.

This document is organized as follows. We begin by giving an overview of a general malware analysis architecture and briefly introduce our specification languages in Chapter 2. The specification languages are then defined in detail in Chapter 3. Finally, we outline current applications within the WOMBAT project of behavioral malware analysis (in Chapter 4) and detection (in Chapter 5). Many of the tools described in these last two chapters have already been adapted to leverage the provided specification languages.

2 General design

2.1 Malware Analysis Architecture

In this section, we give a high-level overview of a generic architecture for the dynamic analysis of malware. The four specification languages defined in this document each play a role in this architecture. We do not focus on any specific analysis tool. In fact, one of the goals of defining specification languages for malware behavior is to provide a common ground between different, independently developed analysis tools. Our architecture is shown in Figure 2.1.

Dynamic malware analysis relies on observing the execution of a malware sample within a controlled environment. This is usually a virtual machine or an emulator that is instrumented to allow a certain amount of information to be logged, such as system calls, API calls, and network traffic generated by the sample under analysis. A malware sample can then be characterized by the sequence of actions it performs. This behavior is then stored in raw, unabridged form in a *Raw Behavior Specification*. This specification (which may be quite large) records all of the information about the malware's execution that the monitoring environment is capable of extracting. The *Raw Behavior Specification* is introduced in Section 2.2.2 and discussed in detail in Section 3.1.

Further analysis builds on top of the *Raw Behavior Specification*. This allows analysis tools to run on recorded executions, without the need to actually execute the malware. Setting up an environment for malware execution is a complex and potentially dangerous task, since malware may require a specific environment to execute. For instance, to operate correctly, a malware may require a specific version of the operating system, or a network connection to a specific server. Furthermore, executing malware is a computationally expensive task, since each malware sample needs to be assigned to a (virtual) host and to run for several minutes. Finally, recorded malware executions can be distributed more freely than malware samples.

The *Behavioral Profile* and the *Abstract Malicious Behavior Language* are intermediate, machine readable formats, suitable as input for a number of analysis tasks such as malware clustering, classification, data mining and behavior-based malware detection. Each of these formats contains a subset of the information available in the corresponding *Raw Behavior Specification*. Such intermediate formats are needed because they

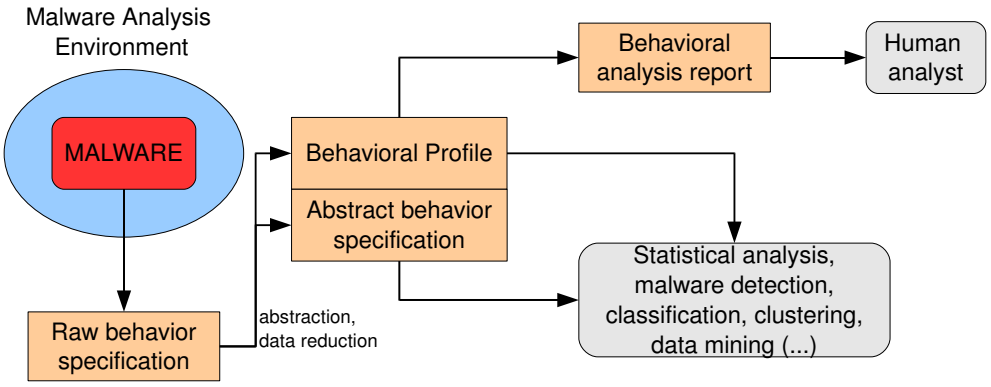

Figure 2.1: Malware analysis architecture

abstract from the fine details of the malware execution that are specific to the execution environment. They also serve the purpose of reducing the size of the recorded malware execution, facilitating both storage and sharing of these traces. Two different languages are needed because they perform different types of abstractions and choose different trade-offs between *abstraction* and *completeness*, and are therefore better suited to different applications. These languages are introduced in Sections 2.2.3 and 2.2.4, and they are described in detail in Sections 3.2 and 3.3.

Finally, the *Behavioral Analysis Report* is an XML format that provides a summary of the malware's behavior which is suitable for a human malware analyst, listing high level behavior such as sending an email, modifying a registry key or downloading and executing a file. This language is introduced in Section 2.2.5 and it is described in detail in 3.4.

2.2 Specification Languages

2.2.1 A Flexible Language Framework

To be widely useful, a specification language for code behavior must be flexible enough to allow malware analysis tools with widely different feature sets to speak a common language. The malware analysis environment shown in Figure 2.1 may be implemented in several different ways, depending on the tool's specific goals and requirements. For instance, a malware detection system designed for production use on end users' computers will need to have a very low impact on a normal usage of the system in terms of reliability and performance. On the other hand, a malware analysis system designed to be run on dedicated hosts by security researchers and organizations has much weaker design constraints. Different constraints lead to different design decisions, which in turn lead to a wide variation in the amount of information on malware execution that can be provided by the analysis environment. As an example, fast logging of system calls through a dedicated kernel driver may be possible on production systems, but using dynamic data tainting [25, 60] to observe data flow inside the malware has an unacceptable overhead in that environment.

Our language design reflects this flexibility, without choosing a "lowest common denominator" approach. None of the information in our recorded malware executions is mandatory. Each malware analysis environment will therefore provide all of the information it is capable of extracting. Clearly, when comparing malware execution traces obtained by different malware analysis environments it will be necessary to take these differences into account.

2.2.2 Raw Behavior Specification

The first language required for a complete architecture for the analysis of malware is a language that is directly suitable as output format for the malware analysis environment. The *Raw Behavior Specification* fills this role. The *Raw Behavior Specification* is a simple yet complete log of all of the analyzed binary's activity that the monitoring environment is able to capture. This includes information on the binary's interactions with the execution environment, such as the system calls and library calls executed by the binary, the parameters with which they are invoked, as well as full network traffic logs. As discussed in Section 3.1, a *Raw Behavior Specification* may also include information on data flows within the monitored binary.

2.2.3 Abstract Malicious Behavior Language

Direct analysis of the raw collected data is unfortunately too cumbersome to understand malware behavior clearly. There is a trade-off between the completeness of information provided by raw data, and the more meaningful but potentially less complete abstracted data. The appropriate level of abstraction depends on the specific goals and use cases.

For detection purposes, the abstracted data must be described by a sufficiently generic language; this is important because it is the only way to prevent malware from evading detection using simple modifications. The explosion of the number of variants from known strains can be seen as a consequence of exceedingly specific detection methods such as binary scanning. Even basic behavioral methods can still be bypassed by basic mutations at the functional level [42]. A high level of abstraction is thus required to cover whole classes of behavior instead of single instances. This level of abstraction can be achieved by getting detached from the specifics of the platform configuration (hardware, operating system, applications) and the language in which the piece of malware has been coded (native, interpreted, macros, ...) [41]. An abstract language satisfying these criteria has been provided in the context of the WOMBAT project and it is described in the dedicated section 3.2. This section also addresses the process of translation into this abstract language with examples from two translation tools developed as proofs of concept: one for PE executables relying on Windows Native APIs and one for Visual Basic Scripts.

2.2.4 Behavioral Profile

A *Behavioral Profile* is also a more concise and abstract representation of code behavior, but it has different goals compared to the *Abstract Malicious Behavior Language* and therefore different trade-offs between generality and completeness. *Behavioral Profiles*

do not attempt to achieve OS or language independence. Rather than malware detection, current applications of this language are related to malware clustering and classification, as well as data mining. A *Behavioral Profile* represents code behavior as a set of OS objects (such as files, or network sockets) and operations on these objects (such as writing to a file), as well as relations between these objects (such as data flow between a file and a socket). We discuss this language in detail in Section 3.3.

2.2.5 Behavioral Analysis Report

A *Behavioral Analysis Report* presents the results of the binary analysis in a human-readable form to the user. The main goal is to provide a compact and yet detailed view on the behavior of a binary. Therefore, the data gathered in previous analysis steps has to be filtered and processed to produce a report that contains information useful to a human reader without omitting relevant information. The resulting *Behavioral Analysis Report* can be used by security analysts to quickly get a complete understanding of the behavior of the binary.

Although the *Behavioral Analysis Report* is created for human users, it is still useful to ensure that it is machine readable, because this provides the means to generate reports that have a different focus without the need to process the analysis data again. Also, the report can be processed and expanded by other applications. The content of this report is explained in detail in Section 3.4

2.3 Usage Scenarios

In the context of the WOMBAT project, a number of tools are being developed with the aim of analyzing or detecting malicious code. In this section, we briefly discuss how these tools can take advantage of our specification languages for code behavior. The tools themselves are further discussed in Chapters 4 and 5.

Raw Behavior Specification. The *Raw Behavior Specification* is now the native output format of the Anubis system [1]. Anubis is a tool for automating the analysis of malicious software that was developed at the Secure Systems Lab of the Technical University Vienna. The main component of Anubis is a full system emulator that is used to run malicious software and produce a log of its behavior. This component can be seen as an implementation of the malware analysis environment shown in Figure 2.1. For more information on Anubis itself, please refer to the dedicated Section 3.2.4 of WOMBAT deliverable *D06 (D3.1) Infrastructure Design.*

The *Raw Behavior Specification* allows sharing of detailed, low level information on malware behavior without sharing the malware sample itself. This information can be used as input for a variety of tools for malware analysis and detection. One example is the S^2A^2DE tool[55, 72] developed at Politecnico di Milano. S^2A^2DE is an anomaly-based intrusion detection system based on system calls. Recent enhancements to S^2A^2DE, developed in the context of the WOMBAT project, are discussed in Section 5.1. S^2A^2DE could be easily modified to perform offline detection on *Raw Behavior Specifications*, using it effectively as an alternate input format. This has not been implemented yet, mainly because of the different target platforms (Anubis analyzes Windows binaries, while S^2A^2DE detects intrusions on UNIX systems). This implies a much broader set of changes in S^2A^2DE, besides implementation of a new input format, before it can be usefully employed on *Raw Behavior Specification* files.

A *Raw Behavior Specification* can also be automatically converted to any of the other three, higher-level specification languages described in this document.

Abstract Malicious Behavior Language. The main goal of the *Abstract Malicious Behavior Language (AMBL)*, is to provide a platform- and language-agnostic framework for the detection of malicious software. Therefore the primary application of this language is malware detection. A system for malware detection that takes AMBL as input has been developed in the context of the WOMBAT project. It is briefly described in Section 5.2.

Behavioral Profile. *Behavioral Profiles* provide a more synthetic but still rich summary of the behavior of analyzed code. These profiles are suitable for being stored in a database in a structured way, so that information on the behavior of all samples in a large collection of malware can be efficiently accessible to the human analyst. We briefly discuss our database of malware behaviors in Section 4.2.

Behavioral Profiles are also the input format for the tool for the behavioral clustering of malware developed by the Secure System Lab of the Technical University Vienna [15]. This tool was developed in the context of the WOMBAT project, and it is described in Section 4.3. Finally, a *Behavioral Profile* can be automatically converted to a human-readable *Malware Analysis Report*.

Malware Analysis Report. *Malware Analysis Reports* provide a simplified, high-level view of the behavior of analyzed code that is suitable for a human analyst. Anubis [1] offers a web service allowing users to submit a code sample for analysis. A *Malware*

Analysis Report is then displayed to the user. The Anubis web service is briefly discussed in Section 4.1.

3 Specification Languages

In this chapter, we describe the different specification languages for malware behavior in detail.

3.1 Raw Behavior Specification

3.1.1 Design goals

Completeness. The *Raw Behavior Specification* is meant to store the *complete* observed behavior of a malware sample for later analysis. Therefore, the language needs to be able to represent all information made available by malware analysis environments. This includes:

- system calls (including parameters)
- library calls (including parameters)
- network traffic (full logs including payloads)
- information flow (as provided by dynamic data tainting instrumentations)

Efficiency. Performance of the malware analysis environments, in terms of the overhead they impose on analyzed software as well as memory footprint, is an important issue. *Raw Behavior Specifications* are meant to be written by the malware analysis environment while analyzing the malware. In some cases, logging of the observed malware behavior can be the performance bottleneck. This means that any increase in the amount of information that is logged would lead to a further slowdown of the binary under analysis. Therefore, the *Raw Behavior Specification* should be as concise as possible, and producing it should not require the malware analysis environment to perform any additional complex computation.

3.1.2 Language

This chapter explains and defines the format of the *Raw Behavior Specification*. The *Raw Behavior Specification* consists of two files:

Network Traffic Dump This file stores the network traffic of the analyzed binary in PCAP format [5]. The PCAP format is is the most accepted format for storing networking traffic today. This way, we have a wide range of standard network analysis tools at our disposal to aid us in our analysis efforts.

Execution Trace This file stores the system calls and library calls performed by the analyzed binary. In addition to the name and argument values of a called function the file also includes information as provided by the analysis environment's dynamic data tainting facility. The format of this file is specified in the following paragraphs.

Execution Trace

One of the first design decisions that we made was to store the behavior of malicious code as a sequence of system and library calls. Naturally, an instruction-trace (a trace of all executed machine instructions) might be more desirable in terms of reaching the completeness design goal. But the storage requirements of an instruction trace are huge while the information gained by having an instruction-trace compared to a function call trace is limited. For this reason, we decided for the function call trace. However, the execution trace's format was designed with extensibility in mind and so it's theoretically possible to include a binary's executed instructions as well.

The execution trace is stored as an ASCII text-file where each line represents an event related to execution of the analyzed binary. Lines are sorted by time so that the last line of an execution trace refers to the last execution event. Each line of the execution trace has the following structure:

```
Line      ::= Timestamp Ws+ Component Ws+ ComponentSpecific
Ws        ::= " "
Timestamp ::= [:asciiCharacter]{15}
Component ::= String
String    ::= [:asciiCharacter]+
```

As one can see each line in the execution-trace starts with a timestamp. The *Timestamp* is a fixed-length string giving the time since the start of the analysis. The potential granularity of the timestamp depends on the analysis environment and its underlying operating system. For this reason, the timestamp precision is not defined here. However, to enable easy parsing of the execution trace the length of the timestamp is defined to be exactly 15 characters. For being of practical use, an analysis environment should make an effort to provide at least timestamps with a granularity of 1/100 second.

The *Component* is a variable-length string consisting of alphanumeric characters and the underscore that describes to which analysis component the event relates. In particular, the component string determines the format of the rest of a line. This feature makes the format of an execution trace extensible. Applications parsing the file simply ignore lines that originate from analysis components that are added at a later time and are thus unknown to the application. Currently, two analysis components are defined: "function" and "compare". Details are provided in the following paragraphs.

The terminal *ComponentSpecific* depends entirely on the specified component. At the moment, only two components ("function", "compare") are specified.

Logging of Functions The execution trace contains one line when a function is called and another line when the function returns. Having two entries per function allows us to correctly display arguments that have a different value at invocation time than at return-time.

More formally, if the component-string either equals "C" or "R" the line is to be interpreted as follows:

```
FunctionCall   ::= TimeStamp Ws+ "C" Ws+ Thread Ws+ FunctionName Ws+
                   "("
                      (ArgName ":" TaintUseSeq? ArgValue)?
                      ("," ArgName ":" TaintUseSeq? ArgValue)*
                   ")"
FunctionReturn ::= TimeStamp Ws+ "R" Ws+ Thread Ws+ FunctionName Ws+
                   "("
                      (ArgName ":" TaintCreateSeq? ArgValue)*
                      ("," ArgName ":" TaintCreateSeq? ArgValue)*
                   ")"
                   (":" TaintCreateSeq? ReturnValue)?
Thread         ::= ProcessName Ws+ Pid Ws+ Tid
ProcessName    ::= QuotedString
Pid            ::= Integer
Tid            ::= Integer
QuotedString   ::= "\"" [:asciiCharacter]+ "\""
Integer        ::= [0-9]+
FunctionName   ::= String
ArgName        ::= String
ArgValue       ::= QuotedString | Integer | FpNumber
ReturnValue    ::= QuotedString | Integer | FpNumber
```

```
TaintUseSeq    ::= "**" TaintUse+ "**"
TaintUse       ::= "<" TaintLabelId ";" FunctionName ";" ArgName ">"
                   NumTainted?
NumTainted     ::= ":" Integer
TaintLabelId   ::= Integer
TaintCreateSeq ::= "**" TaintCreate+ "**"
TaintCreate    ::= "{" TaintLabelId "}"
```

Note that for each logged function activity we specify which process and thread performs the corresponding action. This makes it possible to record the behavior of several processes in the execution trace. This ability is essential because current malware samples are complex programs whose full behavior can only be observed by watching several processes.

The non-terminal *ProcessName* describes the human-readable name of the process that performs the function call or function return. Under process name we understand the name of the executable file whose start resulted in creation of the particular process. Since the process-name might contain spaces and special characters we represent it as a quoted string. A quoted string is enclosed by the character " and allows characters and escape sequences as in the programming language C inside.

The *Pid* is an integer value that represents a unique process-identifier. It is necessary because the same executable file might be started several times resulting in several processes with the same process name. Note that this integer value does not necessarily reflect a specific process-identifier from the analysis environment's guest operating system but can be a counter that is incremented for each different process captured in the execution trace.

The production *QuotedString* describes a string as known from common programming languages. It is enclosed by double quotes and supports the same escape sequences as a string in the programming language C. In particular, one can escape the character " by preceding it with a backslash in order to build strings that contain the " character.

FunctionName gives the name of the function that is called or is returning.

ArgName gives the name of a function's parameter.

ArgValue refers to a parameter's value in an actual function invocation. *ReturnValue* refers to the return value of a function. The representation of an argument or return value is different depending on its type. Since the API of the Windows operating system (but also other systems such as Linux) is defined in header-files of the programming language C we can automatically infer an argument's type by parsing the declarations in the appropriate header files.

Following is a description how each basic type of the C programming language is

represented in the execution trace:

1. `char`, `short`, `integer`, `long` , `enum` arguments (whether signed or unsigned) are printed as numbers.

2. `float`, `double` arguments are printed as floating-point numbers.

3. `char *` arguments are printed as QuotedStrings.

4. `wchar_t *` arguments are converted to ASCII before being printed as QuotedStrings.

5. By default `UserDefinedType *` arguments (i.e. pointers to user-defined types) are represented by printing the pointer-value (i.e. the address) as integer values. It is however possible to add output rules for specific user defined types. It makes sense, for example, to add special code for handling pointers to NT's ubiquitous ObjectAttributes structure `OBJECT_ATTRIBUTES *` or for handling pointers to the various HANDLE-types.

In addition to analyzing function calls we expect our analysis-system to leverage a dynamic data tainting system that makes it possible to track information flows. At least, the execution trace is designed in such a way that allows inclusion of information-flow related data. We assume that the taint-system taints all values returned by a function (i.e. all out arguments and the return value is tainted) with a specific label. Furthermore, we assume that the taint system checks all arguments of a function at invocation time for taint-labels.

Specifically, for each function call line and for each argument the execution-trace can optionally include a *TaintUseSeq*. This piece of information indicates the origin of an argument's value. Of course, the analysis system will not always be able to give this information. However, when present it contains the taint label ID, a value uniquely describing each taint label, the name of the function that returned the value, the name of the argument of that (creator-) function and an optional value (*NumTainted*) specifying how many bytes were tainted with the preceding taint label. Since taint labels might be assigned and propagated with byte granularity an argument/return value consisting of multiple bytes might contain several different taint labels. We accounted for this fact by allowing each argument to list several taint labels. Note that a *TaintUse* label incorporates more information than strictly necessary. Having the taint label ID would be sufficient because the function name and argument name can be found out by searching the execution trace for the *TaintCreate* label with the same taint label ID. Nevertheless,

directly including the function name and argument greatly contribute to the readability of the execution trace.

For each function return line and for each argument or return value the execution-trace can optionally contain a *TaintCreateSeq*. The *TaintCreateSeq* gives the taint label ID or the list of taint label IDs that the analysis environment's taint-system assigned to this argument's value.

Let us conclude the formal description by simply giving an example how the call and return of the `NtClose` function would look like in the execution trace:

```
00:00:07.926724 C "exec.exe" 0 1 NtClose(Handle:
**<808;NtOpenFile;FileHandle>: 4** 1936)
00:00:07.929612 R "exec.exe" 0 1 NtClose(Handle:
1936): **{844}** 0
```

Logging of Comparisons The execution trace is also able to store comparisons with tainted data that take place during the execution of the analyzed binary. If the analysis-system supports such a functionality the execution-trace expects them in the following format:

```
Comparison     ::= TimeStamp Ws+ "T" Ws+ Thread Ws+ CompareName Ws+
                   "("
                      "T0:" TaintUseSeq? ArgValue
                      ", T1:" TaintUseSeq? ArgValue
                   ")"
                   (":" ReturnValue)?
CompareName    ::= "CMP_B" | "CMP_W" | "CMP_L"
```

As one can see comparisons are logged in a format similar to function calls. The difference is that for each comparison only one line exists in the execution trace.

Following is an example of a comparison.

```
00:00:12.295773 T "W3NTSKAA.VXE" 1 1 CMP_B(T0:
**<13824;GetCommandLineA;_functionResult>**, T1: "\t"): 0
```

3.2 Abstract Malicious Behavioral Language

Results from this section have been submitted to the IEEE Symposium on Security and Privacy (SSP'09).

3.2.1 Design goals

Language goals:

- The first goal is the translation of the collected data into a compact, human-understandable representation. This representation must be suitable to feed further analysis processes.
- The second goal is the abstraction from the platform configuration and the programming language. Ultimately, the abstract representation must be interoperable with various collection mechanism such as trace collectors for executables or script analyzers.
- The third goal is a direct consequence of the second goal. Considering any programming language or platform, the development of an automated translation tool must be conceivable. Current developments have already provided two tools for Windows platforms supporting interpretation of PE traces using Windows Native APIs and Visual Basic Scripts. JavaScript is considered for future perspectives.

Language usage:

- The abstract language has been specifically designed for detection as it is described later in Section 5.2. The abstraction brought by translation into the language reduces the risk of evasion by simple modifications of known malware.
- On the other hand, the abstract language is not suitable for analyses which are very sensitive to the data completeness, such as automated learning: during the learning process, the significant information is not identified yet, and it could be lost during translation. It is not suitable, either, for studying malware phylogeny, since technical resemblances are blurred by the abstraction.

3.2.2 Language

Abstract specification of malicious behaviors relies on attribute grammars. Formal grammars are interesting to consider because they are easy to understand and manipulate, but at the same time they provide for sound proofs and automatic analysis. In the context of malicious behaviors, syntactic rules describe the possible combinations of basic operations making up the behavior. The additional semantic rules control both the data flow between the elements involved in these operations, and associate these elements with a potential purpose in the malware lifecycle (installation, communication, execution, ...).

Theoretical framework

From a theoretical perspective, an attribute grammar (Definition 1) is a Context-Free Grammar (CFG) enriched with semantic attributes and rules [46]. In the formalism, each start symbols begins the description of a new malicious behavior. The terminal symbols of the grammar then correspond to the basic operations making up the behavior whereas the production rules describe their different combinations to achieve the behavior. Basic operations eventually refer to the abstract interpretations of the data collected (interpretations of instructions, API calls, arguments) [43, 42].

Definition 1 *An attribute-grammar G_A is a triplet $<G, D, E>$ where:*
- G is originally a context-free grammar $<V, \Sigma, S, P>$,
- let $att : X \in \{V \cup \Sigma\} \rightarrow att(X) \in Att^*$ be an assignment function for attributes and $D = \cup_{\alpha \in Att} D_\alpha$, their set of values,
- E is a set of semantic rules such as for any production rule $\pi \in P$, there is for each variable, at most, one rule $Y_i.\alpha = f(Y_1.\alpha_1 ... Y_n.\alpha_n)$ where $f : D_{\alpha_1} \times ... \times D_{\alpha_n} \rightarrow D_\alpha$.

Overview of the Abstract Malicious Behavior Language (AMBL)

A generic programming language is required to describe any malicious behavior: the Abstract Malicious Behavior Language (AMBL) has been developed to this purpose. By design, the AMBL language focuses on describing the final purpose of a behavior rather than the technical solutions used to achieve it. The high level language can then be declined into more concrete models or instantiations by refinement. Its inner principles are object-oriented as described by the encapsulation of Figure 3.1. The AMBL relies on internal basic operations: arithmetic and control operations guaranteeing Turing completeness, as well as interactions to interface with external objects: commands (open, create, close, delete) or inputs/outputs (send, receive, signal, wait). Malware being resilient and adaptable by nature, interactions are key features.

The atomic operations and interactions constitute the terms of the behavioral language. The syntax of the language constrains the building of these terms and their parameters as given in the production rules (1) to (7) in the Figure 3.2. Using additional production rules, these terms are then sequentially combined into blocks. These blocks can then be combined into parallel, conditional or loop structures making up the behavior. The complete syntax of the language, as well as its operational semantic for execution, are given in [43].

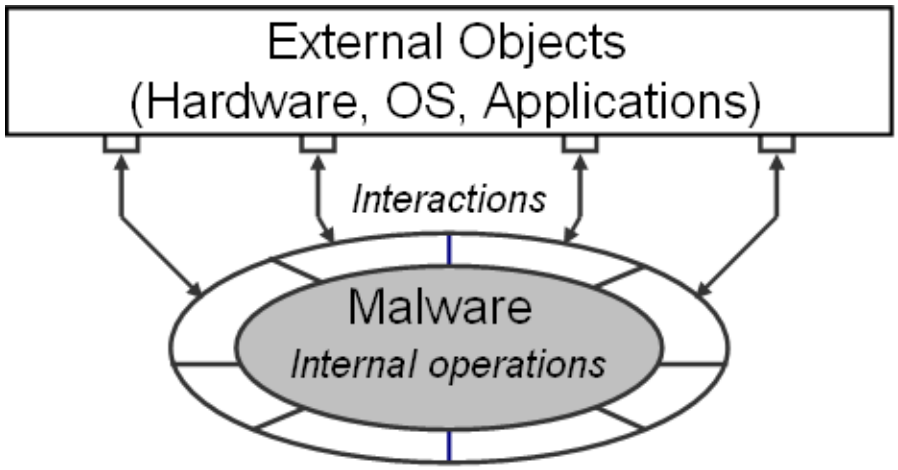

Figure 3.1: Malware object-oriented encapsulation.

(1) <*Op*1>	::= $\neg \mid \&$
(2) <*Op*2>	::= $\vee \mid \wedge \mid \oplus \mid < \mid \leq \mid = \mid \geq \mid > \mid + \mid - \mid \times \mid \div \mid \equiv \mid << \mid >>$
(3) <*Term*>	::= *object* | [<*Term*>] | <*Operation*> | <*Interaction*>
(4) <*Operation*>	::= *object* := (<*Term*>) | [<*Term*>] := (<*Term*>)
	| <*Op*1> (<*Term*>) | <*Op*2> (<*Term*>, <*Term*>)
	| *goto* <*Term*> | *stop*
(5) <*Interaction*>	::= <*Control*> *object* | <*I*/*O*>
(6) <*Control*>	::= *open* | *create* | *close* | *delete*
(7) <*I*/*O*>	::= *receive object* $\leftarrow$ *object* | *receive* [<*Term*>] $\leftarrow$ *object*
	| *send* <*Term*> $\rightarrow$ *object* | *wait object* | *signal object*

Figure 3.2: Syntax of the atomic operations and interactions.

On top of the syntax, semantic attribute and rules enriching attribute-grammars have been provided. These semantic enhancements have two main purposes which are identifying internal and external objects and enforcing a type system for these objects:

Object binding: Object binding identifies the different instances of objects and variables, and guarantees they are coherently used. It is achieved by affecting specific attributes called identifiers to the terminal symbols representing these objects (denoted $objId$ in the semantic rules). In the context of interactions, object binding constrains the data-flow between objects. The data flow is critical in behaviors such as duplication where the malicious code is transferred from the self-reference to a target object.

Object typing: A type attribute can also be affected to a given object (denoted $objTp$). Types are attached to objects according to their potential use. They are critical to distinguish certain malicious purposes such as booting objects in the case of residency or communicating objects in the case of propagation. A description of the different considered types is given in the next paragraph. Additional characterization of the objects can be achieved through additional attributes. For example, an attribute can store the object nature (denoted $objNat$): variable, file, registry key, network socket, mail, etc. Typing may then be refined according to these additional attributes.

With regards to the provided type system, objects are typed according to their potential use in the malware lifecycle. Basically, objects are separated into three classes. The first class of objects gathers the internal variables and constants (var) used by the malware for its internal operations. The second class gathers the permanent objects (obj_perm) which remain persistent after a complete reboot of the system (e.g. files, directories, registry keys) whereas the third class, on the opposite, gathers the temporary objects (obj_temp) existing only for a finite time, as long as the system remains active (e.g. processes, synchronization objects). Particular objects inheriting of these two last classes are defined more specifically:

- The first permanent subclass is made up of the communicating objects (obj_com). These objects constitute communication channels to remote locations or systems. The definition of a communicating object is very wide. Network connections are the most obvious example but transit locations must also be considered: network drives, intranet or peer-to-peer shared directories, removable devices.
- A second permanent subclass gathers the boot objects (obj_boot). These objects provide the malware facilities to automatically execute. Configuration files for the

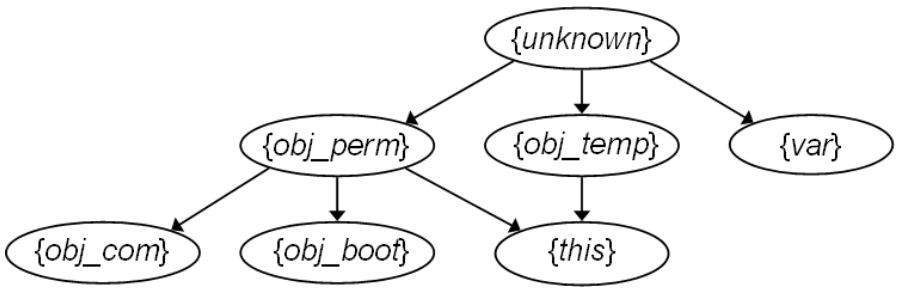

Figure 3.3: Hasse Diagram of the object type poset.

operating system like `win.ini` or `autoexec.bat` for windows, run registry keys, or the Master Boot Record (MBR) are typical means for a malware to be registered in the boot sequence. Automatic execution is also possible at runtime by overwriting the global system service descriptor table, the import tables or entry points in executables. Such locations are also considered as members of the boot class.

- The definition of a self-reference (*this*) shall prove itself useful as in object programming. This element inherits from both permanent and temporary objects since it can be either the drive image of the malware or its associated process in memory.

- In addition to these main classes, additional refinements can still be brought to enrich the typing system. Executable objects (*obj_exe*) can constitute a fourth subclass inheriting from the temporary object. Process and threads are appealing targets for corruption by the malware, in order to gain new privileges for example. An independent class refining permanent objects can also be considered for environment variables (*env_var*). They may be useful in attempts to detect dynamic analysis: redpill techniques for example. Security related objects (*obj_sec*) can finally be built either on environment variables or executables making this subclass hybrid. They play an important role in the protection of the system like antiviral processes or registry keys storing the security configuration for certain web or P2P clients. They may be used by malware for proactive defense.

When enforcing the typing the system, the more specific class always prevails on the generic one. Consequently, a partial order has been defined on these types according to their subset inclusion, as shown in Figure 3.3. In fact, the set inclusions correspond to object specializations.

Descriptions of malicious behaviors

Use of this grammar is best illustrated by examples of behavioral descriptions. In fact, most malicious behaviors can be described by sub-grammars of the AMBL generative grammar. This section only covers two extracts of the most prevalent behaviors: duplication and propagation. Their original descriptions was generated by manually analyzing a pool of malware. Since behavioral descriptions convey only the most generic features of the malicious behaviors, notice that manual generation of these behavioral signatures can be considered easier than generation of binary signatures for virus scanners. In the original paper additional preliminary descriptions can be found for file infection, residency, mutation, overinfection and activity tests [43].

1) Duplication: Duplication is achieved by copying code from the self-reference to a permanent object. The behavior is described below by syntactic production rules (grey) and their related semantic rules (white). The syntactic derivations correspond to the different duplication techniques supported: single-block read/write, interleaved read/write and direct copy with possible permutations. The semantic rules are more interesting. They guarantee the data-flow between the read and write interactions by constraining them to refer to the same variable (Binding: <*Write*>.varId = <*Read*>.varId). They also guarantee the maliciousness of the behavior: the open and read interactions must refer to the self-reference to be a real duplication (Typing: <*Duplicate*>.srcTp = *this*).

(*i*) <*Duplicate*>	::=	<*Create*><*Open*><*Read*><*Write*>	
	|	<*Open*><*Create*><*Read*><*Write*>	
	|	<*Open*><*Read*><*Create*><*Write*>	
	|	<*Open*><*Create*><*InterleavedRW*>	
	|	<*Create*><*Open*><*InterleavedRW*>	
{ <*Duplicate*>.srcId	=	<*Open*>.objId	
<*Read*>.objId	=	<*Duplicate*>.srcId	
<*InterleavedRW*>.obj1Id	=	<*Duplicate*>.srcId	
<*Duplicate*>.srcTp	=	*this*	
<*Open*>.objTp	=	<*Duplicate*>.srcTp	
<*Read*>.objTp	=	<*Duplicate*>.srcTp	
<*InterleavedRW*>.obj1Tp	=	<*Duplicate*>.srcTp	
<*Duplicate*>.targId	=	<*Create*>.objId	
<*Write*>.objId	=	<*Duplicate*>.targId	
<*InterleavedRW*>.obj2Id	=	<*Duplicate*>.targId	
<*Duplicate*>.targTp	=	*obj_perm*	
<*Create*>.objTp	=	<*Duplicate*>.targTp	
<*Write*>.objTp	=	<*Duplicate*>.targTp	
<*InterleavedRW*>.obj2Tp	=	<*Duplicate*>.targTp	
<*Write*>.varId	=	<*Read*>.varId	}

	|	<*DirectCopy*>
{ <*Duplication*>.srcId	=	<*DirectCopy*>.obj1Id
<*Duplication*>.srcTp	=	*this*
<*DirectCopy*>.obj1Tp	=	<*Duplicate*>.srcTp
<*Duplicate*>.targId	=	<*DirectCopy*>.obj2Id
<*Duplicate*>.targTp	=	*obj_perm*
<*DirectCopy*>.obj2Tp	=	<*Duplicate*>.targTp }
(*ii*) <*Create*>	::=	*create object*;
{ <*Create*>.objId	=	*object*.objId
object.objTp	=	<*Create*>.objTp }
(*iii*) <*Open*>	::=	*open object*;
{ <*Open*>.objId	=	*object*.objId
object.objTp	=	<*Open*>.objTp }
(*iv*) <*Read*>	::=	*receive object*1 ← *object*2;
{ <*Read*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Read*>.objId
*object*2.objTp	=	<*Read*>.objTp }
(*v*) <*Write*>	::=	*send object*1 → *object*2;
{ <*Write*>.varId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*2.objId	=	<*Write*>.objId
*object*2.objTp	=	<*Write*>.objTp }
(*vi*) <*InterleavedRW*>	::=	*while*(*receive object*1 ← *object*2;){ *send object*3 → *object*4; }
{ *object*3.objId	=	*object*1.objId
*object*1.objTp	=	*var*
*object*3.objTp	=	*var*
*object*2.objId	=	<*InterleavedRW*>.obj1Id
*object*2.objTp	=	<*InterleavedRW*>.obj1Tp
*object*4.objId	=	<*InterleavedRW*>.obj2Id
*object*4.objTp	=	<*InterleavedRW*>.obj2Tp }
(*vii*) <*DirectCopy*>	::=	*send object*1 → *object*2;
{ <*DirectCopy*>.obj1Id	=	*object*1.objId
*object*1.objTp	=	<*DirectCopy*>.obj1Tp
<*DirectCopy*>.obj2Id	=	*object*2.objId
*object*2.objTp	=	<*DirectCopy*>.obj2Tp }

2) Propagation: Propagation differs from duplication by a different target object: the data is copied from the self-reference to a communicating object. Consequently, propagation shows some syntactic similarities with duplication except readjustments to insert a potential format process: their main differences thus lie in the semantic rules. Two major modifications are brought to the semantic rules of the propagate production. Illustrating the importance of typing, the permanent type of the target object is first replaced by the communicating type (<*Propagate*>.targTp = *obj_com*). A communicating object can either be a network connection, a mail or a file shared over P2P folders and network drives. The second modification specifies, by a disjunction of semantic equations, that the source of propagation can be either the self-reference or the intermediate result of the duplication (<*Propagate*>.srcTp = *this* or <*Propagate*>.srcId = <*Duplicate*>.targId).

(*i*)	<*Propagate*>	::=	<*Open*><*Read*><*Transmit*>
		|	<*Read*><*Open*><*Transmit*>
{	...		
	(<*Propagate*>.srcTp	=	*this*
	∨ <*Propagate*>.srcId	=	<*Duplication*>.targId)
	<*Propagate*>.targTp	=	*obj_com*
	...		}
(*ii*)	<*Transmit*>	::=	<*Format*><*Write*>|<*Write*>

3.2.3 Translation into the abstract specification

Considering raw specification, a trace conveying the actions of a potentially malicious sample is statically or dynamically collected. Depending on the collection mechanism, completeness of the trace data and its nature vary greatly, from simple instructions to system calls along with their parameters. The trace remains specific to a given platform and to the language in which the sample has been coded (native, interpreted, macros). A translation mechanism is thus required to translate the raw data into the abstract behavioral language. Translation of basic instructions, either arithmetic (move, addition, subtraction...) or control related (conditional, jump...), into operations of the language is an obvious mapping which does not require further explanation. However, translation of API calls and their parameters into interactions and objects from the language is more complex and detailed hereafter.

API calls translation

For a program to access any service or resource from its environment, the Application Programming Interfaces (APIs) constitute a mandatory point enforcing security and consistency [62]. API calls may also be denoted system calls when native code is accessing kernel services from the operating system. For consistency reasons, the first notation will prevail. For each programming language, the set of available APIs can be classified into distinct interaction classes. The translation of the API calls is thus mainly language-sensitive. This set of APIs being finite and supposedly stable, the translation can be defined as a mapping over the interaction classes, the completeness of the process being guaranteed. Table 3.1 provides a mapping for subsets of the Windows Native APIs [4] and VB Script APIs. The table is refined according to the nature of the manipulated objects. The API name, on its own, is not always sufficient to determine its interaction class. For example, network devices and simple files use common APIs: for a clear distinction, their path must also be interpreted (`\device\Afd\Endpoint` under Windows). Sending or receiving packets then depends on control codes transmitted with `NtDeviceIoControlFile` (`IOCTL_AFD_RECV`,`IOCTL_AFD_SEND`). If required, specific call param-

Interaction Class	Object Nature	Windows Native API	VBScript API
Open	File	**NtOpenFile**(ptr FileHandle, ..., str FilePath, ...) **NtCreateSection**(ptr SectionHandle, ..., ptr FileHandle)	FileSystemObject.**GetFile**(str FilePath) FileSystemObject.**GetFolder**(str FilePath) FileSystemObject.**OpenTextFile**(str FilePath) FileSystemObject.**FileExists**(str FilePath) FileSystemObject.**GetDrive**(str DivePath) FileSystemObject.**Drives.Item**(int DriveNumber)
	Registry	**NtOpenKey**(ptr KeyHandle, ..., str KeyName, ...) **NtEnumerateKey**(ptr KeyHandle, ...)	
	Network	**NtOpenFile**(ptr DeviceHandle, ..., str NetworkDevicePath, ...)	
Create	File	**NtCreateFile**(ptr FileHandle, ..., str FilePath, ...)	FileSystemObject.**CreateFolder**(str FilePath) FileSystemObject.**CreateTextFile**(str FilePath)
	Registry	**NtCreateKey**(ptr KeyHandle, ..., str KeyName, ...)	
	Network	**NtCreateFile**(ptr FileHandle, ..., str NetworkDevicePath, ...)	
	Mail		**CreateObject("CDO.Message")** **CreateObject("Cdonts.NewMail")** OutlookApplication.**CreateItem**(int ItemNumber)
Close	File	**NtClose**(ptr FileHandle)	FileObject.**Close**()
	Registry	**NtClose**(ptr KeyHandle)	
	Network	**NtClose**(ptr DeviceHandle)	
Delete	File	**NtDeleteFile**(str FilePath)	FileSystemObject.**DeleteFile**(str FilePath) FileSystemObject.**DeleteFolder**(str FilePath)
	Registry	**NtDeleteKey**(ptr KeyHandle)	ShellObject.**RegDelete**(str KeyName)
Read	File	**NtReadFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtReadFileScatter**(ptr FileHandle, ..., ptr SegmentArray, ...) **NtMapViewOfSection**(ptr SectionHandle, ..., ptr BaseAddress, ...)	FileObject.**Read**() FileObject.**ReadLine**() FileObject.**ReadAll**()
	Registry	**NtQueryValueKey**(ptr KeyHandle, str Value, ..., ptr Buffer, ...)	ShellObject.**RegRead**(str KeyName)
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ..., ReadControl, ptr Buffer, ...)	
Write	File	**NtWriteFile**(ptr FileHandle, ..., ptr Buffer, ...) **NtWriteFileGather**(ptr FileHandle, ..., ptr SegmentArray, ...)	FileObject.**Write**(str Value) FileObject.**WriteLine**(str Value) FileObject.**Copy**(str FilePath) FileObject.**Move**(str FilePath) FileSystemObject.**CopyFile**(str FilePath, str FilePath) FileSystemObject.**MoveFile**(str FilePath, str FilePath)
	Registry	**NtSetValueKey**(ptr KeyHandle, str Value, ..., str Buffer, ...)	ShellObject.**RegWrite**(str KeyName,str Value)
	Network	**NtDeviceIoContolFile**(ptr DeviceHandle, ..., SendControl, ptr Buffer, ...)	
	Mail		MailObject.**TextBody**(str Content) MailObject.**Body**(str Content) MailObject.**AddAttachment**(str FilePath) MailObject.**AttachFile.Add**(str FilePath) MailObject.**Attachments.Add**(str FilePath)

Table 3.1: Mapping Windows Native and VBScript APIs to interaction classes.

eters constitute additional mapping inputs:
$\{\text{API name}\} \times (\{\text{Parameters}\} \cup \{\epsilon\}) \rightarrow \{\text{Interaction class}\}$.

Parameters interpretation

Parameters are important factors in interactions, not only to distinguish between ambiguous classes of interactions like previously said but also to identify the different objects involved in interactions and to assess their criticality through typing. Interpretation of the parameters thus complements the initial abstraction from the language obtained through API translation: parameter translation is no longer specific to the language but, rather provides the second step of abstraction from the platform.

Due to their varying nature, parameters can not be translated using a simple mapping. Decision trees are more adaptive tools capable of interpreting parameters according to their representation:

Simple integers: Integer attributes are mainly constants specific to an associated API. They may condition the interpretation of its interaction class. For `NtDeviceIoControlFile`, the different IO control codes are typical examples. A simple hard-coded comparison can detect the main constants.

Address and Handles: Addresses and handles mainly identify the different objects appearing in the collected traces. These parameters are particularly useful to study the data flow between objects. A variable, for example, is represented by an address a_v and a potential size s_v. Every address a such as $a_v \leq a \leq a_v + s_v$ will refer to the same variable. Certain addresses with important properties may be refined by typing: import tables, services descriptor table, entry points. To interpret these specific addresses, a decision tree partitioning the address space is proposed in Figure 3.4.

Character strings: String parameters contain a wide array of information to extract. Most of these parameters are paths satisfying a hierarchical structure where every element is important: from the root identifying drives, drivers and registry, passing by the intermediate directories providing object localization, until the real name of the object. This hierarchical structure is well adapted for a progressive analysis which can be modeled as a decision tree. Figure 3.5 shows a progressive interpretation of the path elements in a Windows configuration.

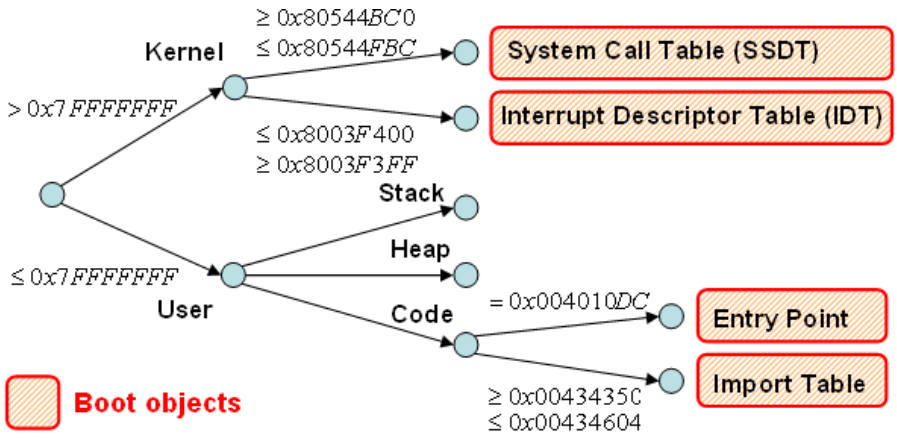

Figure 3.4: Addresses interpretation.

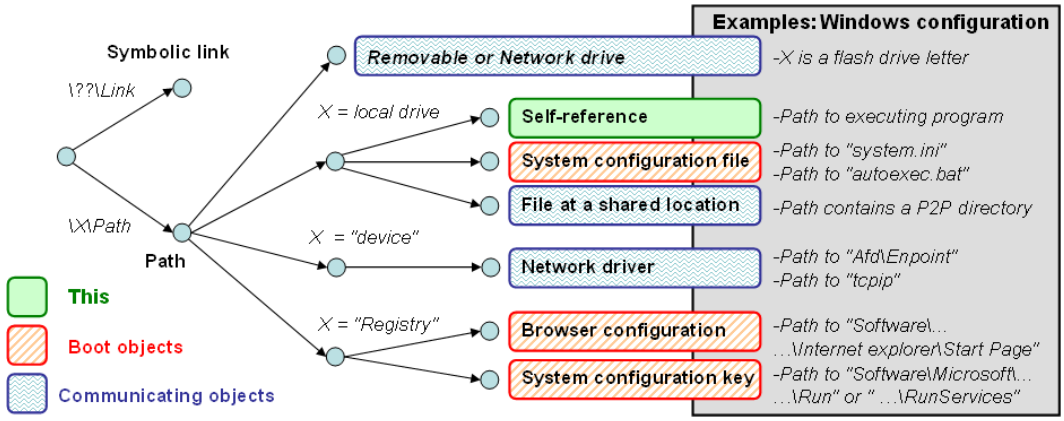

Figure 3.5: Character strings interpretation.

Decision trees generation

Building decision trees requires a precise identification of the critical resources of a system. Our methodology proceeds by successive layers: hardware, operating system and applications. For each layer, we define a scope encompassing the significant components; the resources involved either in the installation, the configuration or the use of these components must then be monitored for potential misuse:

Hardware layer: For the hardware layer, the scope can be restricted to the interfaces open to external locations (Network, CD, USB). The key resources to monitor are the drivers used to communicate with the interfaces. Additional configuration files must also be considered because they may impact the connection parameters (`Host` file) or the booting of the external element (`Autorun.inf`).

Operating system layer: The configuration of the OS is critical but is unfortunately dispersed in various locations (files, registry, tables and structures in memory). The scope is proportionally broadened. However, most of the critical resources are already well identified, such as the boot sequence or the intermediate structures used to access the provided services and resources (file system, process table, system call table...).

Applicative layer: It is obviously impossible to consider all existing applications. To restrict the scope, we limit our analysis to connected and widely deployed applications (web browsers, mail clients, peer-to-peer clients, messaging, IRC clients).

This restriction makes sense since malware have propagation and interoperability requirements to reach large-scale infection. We again consider resources involved in communication (connections, transit locations) as well as in configuration (application launch).

Identification of the critical resources potentially used by malware, is a manual, complex but necessary configuration step. We believe however that this identification is less cumbersome than analyzing the thousands of malware samples discovered every day, for the following reasons. First, critical resources of a given platform are restricted and often known; they can thus be enumerated. Their precise name and location can then be retrieved in a partially automated way. For example, the list of connected drives (local, network, removable media) or the different installed peer-to-peer clients as well as their shared directories can be recovered automatically.

Even though a full automation of the parameter interpretation may be very hard to achieve, anomaly-based intrusion detection already attempted to fully automate the analysis of the system call parameters [49, 55]. Their parameter interpretation relies on deviation measurements from a legitimate model based on string length, character distribution and structural inference. These factors are significant for remote intrusions because these attacks mostly use misformatted parameters to infiltrate systems through vulnerabilities. This approach should prove less efficient with malware, since they mainly use seemingly legitimate parameters. Moreover, these anomaly-based approaches do not explain the use of the object for the malware; this would require an additional manual analysis. Thus, parameter interpretation by decision trees with automated configuration seems a good trade-off between complete automation and manual analysis.

3.3 Behavioral Profile

3.3.1 Design goals

data reduction. *Behavioral Profiles* should be a much more concise representation of a malware's behavior than *Raw Behavioral Profiles*, making them more efficient to store, share and analyze.

abstraction. A *Behavioral Profile* should generalize a *Raw Behavioral Profile*, and represent a malware's execution at a higher level of abstraction.

completeness. A *Behavioral Profile* should still be a complete representation of a malware's execution. All potentially relevant malware behavior should be preserved. This makes this format suitable for a number of applications such as clustering,

machine learning, and data mining. Clearly, there is a trade-off between this requirement and the previous two requirements.

3.3.2 Language

A *Behavioral Profile* is an abstraction of the *Raw Behavior Specification* that contains information about the OS objects that the program operates on, as well as the operations it performs on these objects.

System call traces can vary significantly, even between programs that exhibit the same behavior. For example, consider the different ways to read from a file: Program A might read 256 bytes at once, while program B calls `read` 256 times, reading 1 byte with each call. Moreover, it is easily possible to interleave the read calls with other, independent system calls so that the system call trace changes. For this reason, we abstract the raw behavior specification into a set of operating system objects, together with a set of operations (such as read, write, create) that were performed on these objects.

An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. For example, a behavioral profile might include the file object `C:\Windows` and its accompanying operation `query_directory`. An OS operation is a generalization of a system call that unifies different system calls with similar semantics but different function signatures (e.g., the Windows system calls `NtCreateProcessEx` and `NtCreateProcess` both map to the same operation).

If the raw behavioral specification includes tainting information, it can be used to infer dependences between OS objects. Copying a file, for example, is represented as a dependence between the source file OS object and the destination file object. Dependency information implicitly captures the order of certain operations. This is important, because a behavioral profile does not explicitly consider the order of OS operations that are performed on a specific OS object. The reason is that it should not rely on the order in which unrelated operations are executed. Moreover, dependences can help to determine resource names that are derived from data sources whose values change between execution traces (such as random values or the current time). This information allows the corresponding OS object names to be generalized.

Similarly, if the raw behavioral specification includes information on comparisons involving tainted values (as discussed in Section 3.1.2), these can be translated to comparisons between OS objects.

Specification

As mentioned previously, a behavioral profile captures the operations of a program at a higher level of abstraction. To this end, a sample's behavior is modeled in the form of OS objects, operations that are carried out on these objects, dependences between OS objects and comparisons between OS objects . More formally, a behavioral profile P is defined as an 8-tuple

$$P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$$

where O is the set of all OS objects, OP is the set of all OS operations, $\Gamma \subseteq (O \times OP)$ is a relation assigning one or several operations to each object, and $\Delta \subseteq ((O \times OP) \times (O \times OP))$ represents the set of dependences. CV is the set of all compare operations of type label-value, while CL is the set of all compare operation of type label-label. $\Theta_{CmpValue} \subseteq (CV \times O)$ is a relation assigning label-value compare operations to an OS object. $\Theta_{CmpLabel} \subseteq (CL \times O \times O)$ is a relation assigning label-label compare operations to the two appropriate OS objects.

OS Objects. An OS object represents a resource, such as a file or registry key, that can be manipulated and queried via system calls. Formally, an OS object is a tuple of the following form:

```
OS Object ::= (type, object-name)
type ::= file|registry|process|job|network|thread|section|driver|sync|service|random|time|info
```

That is, an OS object has a name and a type that together uniquely identify the object in the operating system. The 'file' type covers file, named pipe, and mailslot resources, 'registry' consists of registry keys, 'process' includes processes, and 'job' denotes Windows NT jobs, which allow for combining individual processes into a group. The 'network' category describes network objects, 'thread' represents thread activity, 'section' refers to memory-mapped files, and 'driver' captures the loading and unloading of Windows device drivers. The type 'sync' abstracts all synchronization activities, such as operations on semaphores and mutexes, and 'service' contains objects that represent Windows services. The type 'random' includes several sources of randomness, each of which can be used by a program to generate a random number. The type 'time' consists of time sources, and 'info' contains only two objects. One is the object *info-executable*, which represents the loaded executable. The other one is *info-general*, which represents information such as pathnames of the windows system directory and the temporary directory.

To create OS objects, the raw behavioral specification can be scanned for all system calls that produce new OS resources. For example, the function `NtCreateFile` creates

OS Object		*OS Operation*	
Type	*Name*	*Name*	*Attributes*
net	http_server	contact	‘www.gasolution.com‘, ‘80‘
net	http_request	get	‘/downloader/start2.htm‘
net	dns_resolver	query	‘Type A‘, ‘mx0.gmx.net‘
net	port_listener	listen_on	‘TCP‘, ‘6777‘
net	smtp_attmts	send	‘fpw.exe‘

Table 3.2: Example network OS objects.

new files. For each such system call, the object name can be extracted from the argument list, while the object type can be deduced from the type of the system call. Typically, native API calls have a parameter, named `ObjectAttributes`, that can be directly translated to an object name. In a few cases, it is more difficult to determine the object name. For example, `NtCreateProcess` expects a handle argument that points to a section object (a memory-mapped file), instead of an argument that specifies the filename of the executable. To address this problem, we have extended our system call logger to resolve handles to NT kernel objects and provide this information.

As discussed in Section 3.1.2, the malware analysis environment may also monitor network activity at the network level, by using a sniffer to capture and network traffic generated by the malware. Raw network traces can be analyzed, extracting network OS objects. Depending on the type of network traffic observed, different kinds of network objects are created. Table 3.2 lists some example network objects, together with their corresponding operations.

OS Operations. An OS operation is a generalization of a system call. Formally, an operation is defined as:

```
OS operation ::= (operation-name, operation-attributes?, successful?)
```

An operation must have a name, it may have one or more attributes that provide additional information about the operation, and it may have a value describing whether the operation was successful.

We map system calls to OS operations with the intent of abstracting from API-specific details. For example, we ignore whether a process is created by means of `NtCreateProcess` or `NtCreateProcessEx` and unify these two system calls into the single OS operation `create`. Our mapping function only considers the most essential system calls, such as functions for reading, writing, and creating operating system objects. This allows

us to abstract from many unimportant details. For example, we ignore all functions relating to NT's Local Procedure Call functionality, because this is an undocumented feature that is not available via the Windows API. Currently, we map 130 native API and Windows API functions to 55 OS operations.

System calls that operate on a resource typically have a (handle) parameter that references the target resource. This is necessary for the OS to know the resource to which an operation should be applied. We make use of these handles to map operations to the appropriate OS objects. After assigning operations to OS objects, all of an object's operations are stored in a set. As a consequence, the order of OS operations is irrelevant. This is important, because it is very easy to reorder system calls on a resource without changing the semantics of a program. Thus, we are able to generalize our behavioral profile by neglecting the order of operations. System call dependences may be used to capture the order between those OS operations where the actual order is implied by a data dependence. Moreover, the number of operations on a certain resource does not matter in our system. This sacrifices some precision, but makes the behavioral profile more general, and thus, harder to evade by introducing superfluous operations.

Object Dependences. We abstract dependences between system calls to dependences between OS objects. While a system call dependence is a dependence relation between two system call instances, an OS object dependence is a dependence between two OS objects and their operations. For each existing system call dependence, we first check whether the two involved system calls map to OS operations. If this is the case, we introduce an object dependence between the corresponding OS objects. Note that this information is only available if the raw behavior specification provide taint tracing information.

Based on this representation of objects and their dependences, it is straightforward to find execution-specific artifacts. For example, we recognize random or temporary filenames by checking whether there is a dependence between a file object and a random source. If this is the case, we do not want to keep the actual object in the profile, since it is different for each execution. Thus, we replace the concrete object name with a placeholder token that indicates the source of the object name (such as TEMPORARY for a temporary filename). Moreover, we append the value of a counter that is increased by one until the object name becomes unique in this profile. When comparing two behavioral profiles that both contain objects with temporary filenames, it is possible to match these two objects. However, we have to avoid that an object a_1 of profile A matches with object b_1 of profile B, when the operations associated with the object make it actually more similar to object b_2 of profile B. We address this problem by calculating a checksum over all OS operations, using the resulting value as part of the

new object name. That is, execution-specific names are replaced with a new name of the form `<token><checksum><counter>`. The checksum guarantees that only objects with the same OS operations will receive the same name in two different profiles, and consequently match.

Control Flow Dependences. Control flow dependences are translated into comparisons between OS objects. Depending on the type of the comparison, a control flow dependency is associated with either one or two OS objects. A label-label comparison involves two OS objects (one for each operand), while a label-value comparison involves only a single one. To find the appropriate OS resource, the labels are use. That is, we search for the OS operation that created a particular label. Then, we search for the object that the operation is associated with.

Example behavioral profile. Figure 3.6 shows an example of a behavioral profile. Note that although this example is shown in C code, our profile extraction algorithm works on execution traces. This example shows code that copies the file `C:\sample.exe` to `C:\Windows\sample.exe` by memory-mapping the source file. As one can see, independent of the number of times the write operation in Line 14 is executed, the write operation appears only once in the corresponding behavioral profile. It is also noteworthy that the `NtQueryAttributesFile` operation in Line 6 is assigned to the object `C:\Windows\sample.exe`, although it does not use a handle argument to reference its OS object. The behavioral profile also contains a dependence between the section OS object of the source file and the file object of the destination file. This dependency reflects the fact that data from the source was copied to the destination file.

3.4 Behavioral Analysis Report

3.4.1 Design goals

human-readable. The *Behavioral Analysis Report* is intended for human users and provides a high-level view on the data that is gathered in the malware analysis. Therefore, the analysis data has to be processed so that information that is relevant to users can be extracted and presented in a form that allows a user to easily obtain an extensive picture of the behavior of a binary.

machine-readable. Although the main goal is to create an human-readable output, the report should still be suitable for automatic processing. This provides the flexibility to generate different kinds of reports and it permits the processing of a report by different programs. In this way, a report can be extended with analysis data from other sources than the one who initially created the report.

```
0: // open the source-file as a memory-mapped file
1:  HANDLE src = NtOpenFile("C:\sample.exe");
2:  HANDLE sectionHandle = NtCreateSection(src);
3:  void *base = NtMapViewOfSection(sectionHandle);
4:
5:  // don't overwrite the target
6:  if (NtQueryAttributesFile("C:\Windows\sample.exe") !=
7:     STATUS_OBJECT_NAME_NOT_FOUND)
8:   exit(1);
9:  // open the target
10:  target = NtCreateFile("C:\Windows\sample.exe");
11:
12:  void *p = base;
13:  while(p < base + fileLen) {
14:    NtWriteFile(target, p++);
15:  }
```

Pseudo Code Fragment

```
File|C:\sample.exe
        open:1
Section|C:\sample.exe
        open:1, map:1, mem_read: 1
File|C:\Windows\sample.exe
        query_file:0, create:1, write:1
Section|C:\sample.exe -> File|C:\Windows\sample.exe
        mem_read – write: (fileLen)
```

Behavioral Profile

Figure 3.6: Example Behavioral Profile

3.4.2 Language

XML was chosen as storage format to represent the *Behavioral Analysis Report*. XML allows both design goals to be met, because files in this format are machine readable and they can easily be transformed into a human readable representation. Another advantage is that there are a lot of implementations for XML parsers in many different programming languages, which allows for a wide variety of tools to create and extend the analysis report. Furthermore, the *Behavioral Analysis Report* language is formally defined by an XML Schema [2]. A major benefit of an XML schema is the fact that the report file can be checked for compliance against it, therefore it can be assured that the generated report is always semantically correct. Once the XML report file is created, it can be transformed into a format that is human readable, e.g. HTML, PDF or plain text.

The content of the XML file that contains the report can be grouped into two major categories: every report contains general information about itself and detailed information about the executable. The general information section includes data such as the creation date of the report, or the total execution time of the binary. The information about the executable is more comprehensive and contains the following sections:

General executable information. This category describes the command line to start the executable, the exit status, the dll's that are loaded, the virus scanner output and the pop-ups that are opened by the binary.

Registry activities. Describes the registry keys that are accessed or modified.

File activities. Shows information about files that are created, deleted or modified by the binary.

Service activities. Reports all interaction with services on the analysis system such as starting, stopping, controlling or deleting a service.

Process activities. Shows information about processes and threads that are created or deleted by the binary.

Network activities. The entire network behavior of a process is recorded in this section. This includes data that is sent across communication channels as well as creation and manipulation of communication endpoints on the system.

Miscellaneous activities. This section records if a driver is loaded or unloaded by the binary, if a mutex is created or if an exception occurred.

Evasion. Reports if the binary tries to evade analysis. Malware can evade automatic behavioral analysis by not performing it's intended malicious function when it detects it is being run inside a malware analysis environment.

A complete formal definition of the *Behavioral Analysis Report* specification language is provided as an XML Schema, and is available from [2].

The main part of the report itself is crafted out of the corresponding *Behavioral Profile*. This *Behavioral Profile* is scanned by a script that extracts information from relevant OS interactions. For example, every action that creates a new file is processed, leading to an entry in the report that displays what file is created. Also, some interactions can be combined into a single action, e.g. when a file is read, the previous calls that are required to open a file do not need to be mentioned in the report.

Optionally, the *Behavioral Analysis Report* may be enhanced by including a full network traffic dump. The reason is that a malware analysis system may not be able to extract all relevant information out of raw network traffic. On the other hand, a variety of existing tools are available that can assist the human analyst in examining a raw network traffic dump. This includes packet dissectors such as Wireshark [71], intrusion detection systems such as snort [68], as well as specialized tools aimed at the analysis of specific network protocols. Therefore, the full dump is provided for later analysis, encoded as a pcap file [5]. Since full network dumps are not available as part of a *Behavioral Profile*, they must be extracted from the *Raw Behavioral Specification*, if it is available.

4 Behavioral Malware Analysis

4.1 Malware Analysis Service

The Anubis service is a *Malware Analysis Service* that analyzes unknown windows binaries and websites. A binary that is submitted to Anubis is executed in an emulated environment where all its actions are monitored. The Anubis web site [1] provides the means to upload binaries and to view reports. Users can submit samples by uploading binaries in a web form, or they can enter a URL that points to a potentially malicious web site. The data provided by the user is processed and analyzed by Anubis, and a *Behavioral Analysis Report* that describes the results of this analysis is then presented to the user. The Anubis tool itself is discussed in WOMBAT Deliverable "D06 (D3.1) Infrastructure Design". Here, we briefly introduce the web interface that makes Anubis behavioral malware analysis available to WOMBAT partners as well as to the public.

There are a lot of submissions to Anubis each day (about 800 thousand in total in 2008, up from 330 thousand in 2007), so the uploaded binary is queued and processed when the required resources are available. A user can prioritize her binary by entering a CAPTCHA code on the submission page so that it gets processed immediately. Anubis also supports the automatic submission of binaries. A special POST request can be used to send the analysis subject to the Anubis service. Afterwards, the URL of the analysis report is mailed to the provider of the binary. The automatic submission can also be performed with the help of a Python script that is available for download on the Anubis web site.

While the Anubis service itself is not a product of the WOMBAT project (the service has in fact been online since March 2007), Anubis is undergoing constant enhancements. This includes hardware upgrades that allow us to analyze more malware samples and usability features that make the service more attractive to end users, and therefore increase the amount of malware samples that are submitted. Other improvements include integration with the WOMBAT framework. The Anubis tool now presents the user with *Behavioral Analysis Reports* that comply to the specification described in this document. Additionally, Anubis now makes use of the *Raw Behavior Specification* and *Behavioral Profile* formats, although reports in these formats are not currently accessible to users of the web interface.

Finally, and perhaps more importantly, the Anubis web service provides a great opportunity for real-world testing and deployment of research developed within the WOMBAT project. As a first example of how this opportunity is being leveraged, work is currently under way to integrate the scalable behavioral clustering described in Section 4.3 into the Anubis web interface. This will provide a real world test of the scalability and effectiveness of this novel tool for malware classification. At the same time, it will provide users of the Anubis service with extremely useful information. In addition to the report on the submitted malware's behavior, the user will be presented with the cluster this sample has been assigned to, and he will be able to browse the reports on other samples in the cluster. An initial version of this functionality is planned to be put online in the first months of 2009.

From the point of view of usability, several enhancements have been made. *Behavioral Analysis Reports* are now made available in a number of formats, in addition to the native XML. Namely, PDF, plain text, and MIME encoded HTML (MHT). There is also a new advanced submission web page that permits the submission of zipped binaries and additional files that are needed by the analysis subject. It also offers the choice between getting the report displayed in the browser once the analysis is finished or receiving an URL pointing to the report by email. Furthermore, Anubis gives a preliminary estimation of how dangerous a binary is. This is illustrated by a traffic light sign, the three different colors indicating how dangerous a binary is. Green means that the binary is most likely not malicious, yellow stands for potentially dangerous and red means that this binary is malicious. Finally, Anubis can now analyze malicious web sites, by automatically visiting sites at submitted URLs with internet explorer.

4.2 Malware Behavior Database

The Malware Behavior Database is a database supporting large-scale storage of *Behavioral Profiles* and *Behavioral Analysis Reports* generated by Anubis analysis in such a way that they can be easily and efficiently searched. This database is currently under development at the Technical University Vienna and Institut Eurecom. For example, this database makes it possible to search for all malware samples that, during analysis, created a file with a particular filename, or communicated with the IP address or domain name of a known botnet C&C server. Moreover, the database will help in providing a global view on the malware landscape. In particular, it will provide the basis for mining of meaningful information out of the wealth of data gathered by Anubis analysis.

We will describe the Malware Behavior Database, toghether with the insight it will provide us on the behavior of malicious software, in WOMBAT Deliverable D16 (D4.2)

"Analysis report of behavioral features".

4.3 Malware Clustering

Results from this section have been accepted for publication in the Symposium on Network and Distributed System Security (NDSS) [15]

Automating the analysis of the behavior of a single malware sample is a first step, but it is not sufficient. The reason is that the analyst is now facing thousands of reports every day that need to be examined. Thus, there is a need to prioritize these reports and guide an analyst in the selection of those samples that require most attention. One approach to process reports is to cluster them into sets of malware that exhibit similar behavior. The ability to automatically and effectively cluster analyzed malware samples into families with similar characteristics is beneficial for the following reasons: First, every time a new malware sample is found in the wild, an analyst can quickly determine whether it is a new malware instance or a variant of a well-known family. Moreover, given sets of malware samples that belong to different malware families, it becomes significantly easier to derive generalized signatures, implement removal procedures, and create new mitigation strategies that work for a whole class of programs.

Grouping individual malware samples into malware families is not a new idea, and clustering and classification methods have already been proposed previously [13, 31, 47, 51, 39]. These approaches, however, generally do not scale well and are too slow for the size of malware sets that anti-malware companies are confronted with. Moreover, these techniques are imprecise, either because their notion of similarity is not tied to a program's actual behavior or because it does not capture a program's behavior well enough. Imprecise in this context either means putting samples of different types into the same group or failing to recognize similar malware programs.

In this section, we present a novel clustering technique that scales well and produces more precise results than previous approaches. This technique is based on analyzing malware behavior. Unlike many previous systems that operate directly on low-level data such as system call traces, our clustering technique takes as input the behavioral profiles described in Section 3.3. This allows our system to recognize similar behaviors among samples whose low-level traces appear very different. Furthermore, our system is designed to be *scalable*, to be able to cluster large, real-world malware datasets such as those gathered by the Anubis service, by SGNET, and by other malware collection efforts within the WOMBAT project.

Clustering a set of n points in a high-dimensional space is a computationally expensive

task. Most clustering algorithms require to compute the distances between all pairs of points in the set. In this case, computational complexity is at least $O(n^2)$ evaluations of the distance function, which is unacceptable for large data sets.

There exist algorithms, such as the k-means algorithm (Lloyd's algorithm) [54], that only compute the distance from the n points to k cluster centers, and repeat this computation for each of i iterations required to converge to a local optimum. The computational complexity is, therefore, $O(nki)$ evaluations of the distance functions. Unfortunately, there are no guarantees that the value of i is small (in fact, the number of iterations is super-polynomial in n in the worst-case [12]). Furthermore, the accuracy of k-means is limited (the solution is only locally optimal), and the number of clusters k has to be specified *a priori*.

In this work, we employ locality sensitive hashing (LSH), introduced by Indyk and Motwani [40], to compute an approximate clustering of our data set that requires significantly less than n^2 distance computations. Our clustering algorithm takes as input the set of malware samples $A = a_1, .., a_n$, where $a_i \subseteq F$, and F is the set of all features. LSH algorithms have been proposed for metric spaces where the similarity between two points is defined by one of a few simple functions, such as Jaccard index [18], or cosine similarity [22] In this work we employ the Jaccard index as a measure of similarity between two samples a and b, defined as $J(a, b) = |a \cap b|/|a \cup b|$. A similarity value of $J(a, b) = 1$ indicates that two samples have identical behavior. While other, more complex similarity functions, such as normalized compression distance [13], may be more accurate measures of the similarity between behavioral profiles, choosing this simple set similarity measure allows our clustering approach to leverage LSH and to scale up to the size of real-world malware collections.

We now describe how we map a behavioral profile into a set of features that are suitable for LSH. Section 4.3.1 briefly explains the LSH algorithm. In Section 4.3.2, we discuss how we can use the output of the LSH algorithm to compute an approximate, hierarchical clustering of a set of malware samples. Finally, in Section 4.3.3 we discuss the asymptotic performace of our approach.

Transforming Profiles into Features Sets Before we can run the clustering algorithm, we have to transform each behavioral profile into a feature set. Informally, a feature is a behavioral characteristic of a sample, such as "file xy was created." We use the following algorithm to transform a behavioral profile $P = (O, OP, \Gamma, \Delta, CV, CL, \Theta_{CmpValue}, \Theta_{CmpLabel})$ into a set of features: For each object $o_i \in O$, and for each assigned $op_j \in OP|(o_i, a) \in \Gamma$, create a feature:

$$f_{ij} = "op|" + name(o_i) + "|" + name(op_j)$$

where $name()$ is a function that returns the name of an OS object, operation, or comparison as string, quotes (") denote a literal string, and + concatenates two strings. Moreover, for each dependence $\delta_i \in \Delta = ((o_{i1}, op_{i1}), (o_{i2}, op_{i2}))$, we create a feature:

$$f_i = "dep|" + name(o_{i1}) + "|" + name(op_{i1}) +$$
$$+" \rightarrow " + name(o_{i2}) + "|" + name(o_{i2})$$

For each label-value comparison $\theta_i \in \Theta_{CmpValue} = (cmp, o)$, we create a feature:

$$f_i = "cmp\_value|" + name(o) + "|" + name(cmp)$$

For each label-label comparison $\theta_i \in \Theta_{CmpLabel} = (cmp, o_1, o_2)$, we create a feature:

$$f_i = "cmp\_label|" + name(o_1) +$$
$$+" \rightarrow " + name(o_2) + "|" + name(cmp)$$

The output of this transformation step is a set of features that captures the behavioral characteristics of a sample in a form that is suitable for the clustering algorithm. We then discard all features of a sample that are unique with regards to all other samples in the data set. That is, we do not consider a feature for clustering when it does not occur in at least one other sample's feature set. This is because a unique feature of a sample does not help us to find other samples that behave similarly (i.e., the information gain of this feature is very low). Moreover, our experiments show that the robustness of our clustering to the selection of the threshold t improves when we discard such unique outliers.

4.3.1 Locality Sensitive Hashing (LSH)

The idea behind locality sensitive hashing is to hash a set A of points in such a way that near (or similar) points have a much higher collision probability than points that are distant. We achieve this by employing a family H of hash functions such that $Pr[h(a) = h(b)] = similarity(a, b)$, for a, b points in our feature space, and h chosen uniformly at random from H. By defining the locality sensitive hash of a as $lsh(a) = h_1(a), .., h_k(a)$, with k hash functions chosen independently and uniformly at random from H, we then have $Pr[lsh(a) = lsh(b)] = similarity(a, b)^k$.

In the case of sets for which the Jaccard index is used as similarity measure, a family of hash functions H with the desired property has been introduced in [18]. A hash in H imposes a random order on the set of all features. The hash value for a feature set a is then determined by the index of the smallest element of a according to this order.

Since it is inefficient to generate truly random permutations, random linear functions in the form $h(x) = c_1x + c_2 \mod P$ are used instead [36], with P a prime number larger than the total number of features in F.

Given a similarity threshold t, we employ the LSH algorithm to compute a set S which approximates the set T of all near pairs in $A \times A$, defined as $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Given the threshold t, we first choose the number k of hash functions in each LSH hash, and the number of iterations l. Furthermore, we initialize the set S of candidate near pairs to the empty set. Then, for each iteration, the following steps are performed:

- choose k hash functions $h_1, .., h_k$ at random from H
- compute $lsh(a) = h_1(a), .., h_k(a)$ for each $a \in A$
- sort the samples based on their LSH hashes
- add all pairs of samples with identical LSH hashes to S

LSH Parameters. For a given similarity threshold t, we must choose appropriate values of k and l. For a pair $p = (a, b)$ such that $similarity(a, b) = v$, we have $Pr[p \in S] = 1 - (1 - v^k)^l = g(v)$. Thus, given t, we can choose k and l such that $g(t)$ is close to 1 and $g(t/(1 + \epsilon))$ is small, for any $\epsilon > 0$. That is, t is the only parameter that needs to be chosen. For a threshold value of $t = 0.7$ we selected $k = 10$ and $l = 90$.

4.3.2 Hierarchical Clustering

The result of the locality sensitive hashing step is a set S, which is an approximation of the true set of all near pairs $T = \{(a, b)|a, b \in A, J(a, b) > t\}$. Because LSH only computes an approximation, S might contain pairs of samples that are not similar. To remove those, for each pair a, b in S, we compute the similarity $J(a, b)$ and discard the pair if $J(a, b) < t$. Then, we sort the remaining pairs by similarity. This allows to produce an approximate, single-linkage hierarchical clustering [53] of A, up to the threshold value t. Single-linkage clustering allows us to simply iterate over the sorted list of pairs to produce an agglomerative clustering. We stop the clustering when there are no more near pairs left.

In some cases, one would like to continue the hierarchical clustering process until all elements are merged into a single cluster. However, all subsequent clustering steps would require to merge two clusters that have a similarity value below t. Of course, this information is not readily available. The reason is that the LSH algorithm avoids the

calculation of distances between elements that have a similarity value below t. To solve this problem and to obtain an exhaustive, hierarchical clustering, we use the following technique: We choose a representative element for each cluster, calculate the distances between all representatives, and then perform exact, hierarchical clustering between these elements. We create the representative element r of a cluster C by adding all features to r_C that exist in at least half of all the feature sets in C. Of course, exact hierarchical clustering has a complexity of $O(n^2)$. This is acceptable because the number of representatives is very low.

4.3.3 Asymptotic Performance

The LSH scheme described previously requires the computation of nkl hashes. The computational complexity of each hash of a sample a is $O(|a|)$. Therefore, the overall complexity of the hashing step is $O(ndkl)$, where $d = avg(|a|), \quad a \in A$, is the average number of features in a sample. After hashing, $|S|$ similarity functions must be computed.

The set S is an approximation of the true set of all near pairs T. We may, therefore, have false negatives $(T - S)$, and false positives $(S - T)$. We have $|S| \leq |T| + |S - T|$. Clearly, $|T| < nc$, where c is the maximum cluster size for the given threshold. Unfortunately, we cannot provide a theoretical bound for the fraction of false positives $|S-T|/|S|$ without making some assumptions on the distribution of the distances between pairs in A. However, in practice, the value is small (below 0.19 in our experiments). Therefore, the number of similarity computations is limited by the size of $|T|$ and the complexity of $O(nc)$. Since a single similarity computation is $O(d)$, computational complexity of this step is $O(ncd)$. Finally, the pairs in S need to be sorted to perform hierarchical clustering. This step is $O(nc \log(nc))$.

For large data sets, the cost of the similarity computations, which is $O(ncd)$, dominates. Note that while in practice nc is significantly smaller than n^2, the asymptotic performance has not improved. The reason is that c can still be $O(n)$ in the worst case. Consider for instance a trivial dataset where all n samples are identical. Clearly, for such a dataset we would have a single cluster of size n (and therefore $c = n$) for any t. More generally, if the threshold value t is too high it may lead to most samples being concentrated in a few large clusters. However, for reasonable values of t, the performance gained by using LSH is sufficient to allow us to cluster large, real-world malware data sets.

For extremely large datasets, on the other hand, more aggressive approximate clustering techniques may need to be employed (at the cost of some accuracy), such as the ones described in [36]. In [36], LSH is used to generate the set of approximate near pairs

$|S|$, but there are no similarity computations. A pair $(a, b) \in S$ is not verified to be near by computing $similarity(a, b)$, but by using a faster approximate method based on the already computed hashes.

5 Behavioral Malware Detection

5.1 System call anomaly detection using sequence and parameters

Results from this section have been partially published in ACM Operating Systems Review ad F. Maggi, S. Zanero, V. Iozzo: "Seeing the Invisible - Forensic Uses of Anomaly Detection and Machine Learning"

Most of the anomalous actions that an aggressor would try to perform on a system through uploaded malware (e.g., accessing the host file system, sending or receiving packets over the network, executing other programs on the host, etc.) require the use of one or more system calls. Thus, it is reasonable to monitor such calls in order to analyze the behavior of a process. In particular, we propose to use anomaly detection techniques to flag anomalous or suspicious executions and record them for review in order to create a trail (i.e., the alert logs) that would otherwise be lost. We use S^2A^2DE, a tool which we developed in [55, 72], which makes use of both the sequence and the content of system calls to detect anomalies. This has been shown to be more efficient than using sequences of syscalls only, something which has been studied for a long time since the seminal work [26].

S^2A^2DE is a next-generation evolution of the seminal works in the field by Vigna et al. [50, 57], and uses a Markovian model of the sequence (as in, e.g., [48]) complemented with an analysis of the arguments of the system calls to detect intrusions - and malware activity.

The architecture of S^2A^2DE is shown in Figure 5.1. Each execution of an application is modeled as a sequence of system calls, $S = [s_1, s_2, s_3, \ldots]$, logged by the operating system auditing facilities. Each system call s_i is characterized by a *type* (e.g. `read`, `write`, `exec`, etc.), a list of *arguments* (e.g., the path of the file to be opened by `open`), a *return value*, and a *timestamp*. The return value is not taken into account, neither the *absolute* timestamp (the sequence of the system calls is considered instead).

S^2A^2DE must be trained in order to "learn" a model of the normal behavior of the monitored applications. During this phase, the system builds a distinct profile for each application (e.g. `sendmail`, `telnetd`, etc.). A two-phase process of machine learning is

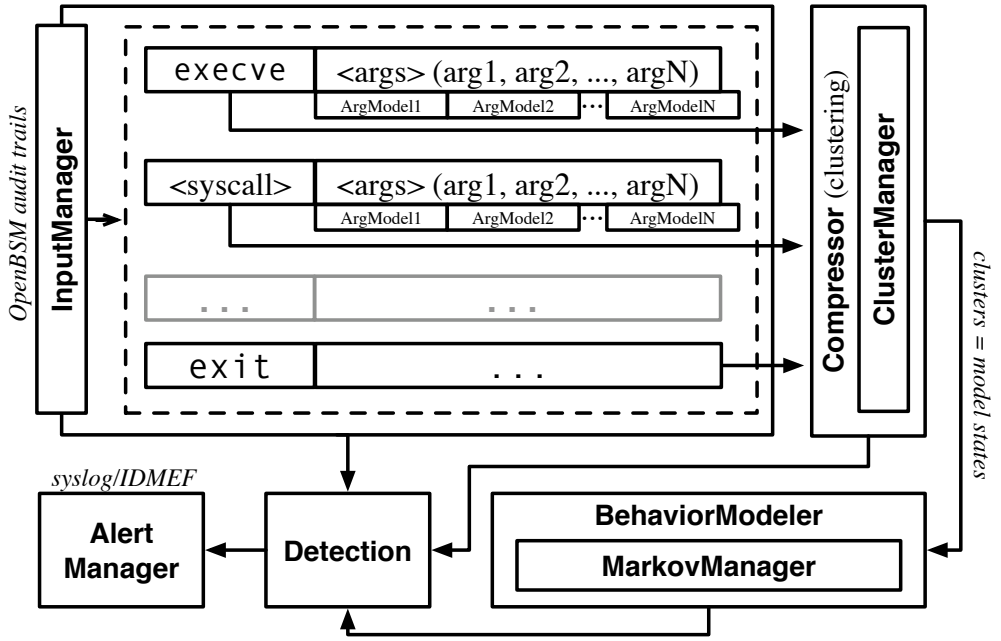

Figure 5.1: The architecture of our HIDS prototype

Table 5.1: Association of models to Syscall arguments in our prototype

Syscall	Model used for the arguments
`open`	`pathname` → Path Name `flags, mode` → Discrete Numeric
`execve`	`filename` → Path Name `argv` → Execution Argument
`setuid, setgid`	`uid, gid` → User/Group
`setreuid, setregid`	`ruid, euid` → User/Group
`setresuid, setresgid`	`ruid, euid, suid` → User/Group
`symlink, link,rename`	`oldpath,newpath` → Path Name
`mount`	`source, target` → Path Name `flags` → Discrete Numeric
`umount`	`target,flags` → Path Name
`exit`	`status` → Discrete Numeric
`chown` `lchown`	`path` → Path Name `group, owner` → User/Group
`chmod, mkdir` `creat`	`path` → Path Name `mode` → Discrete Numeric
`mknode`	`pathname` → Path Name `mode, dev` → Discrete Numeric
`unlink, rmdir`	`pathname` → Path Name

then applied to each type of system call separately. Firstly, a single-linkage, bottom-up agglomerative hierarchical clustering algorithm [33] is used to find, for each type of system call, sub-clusters of invocations with similar arguments. We are interested in creating models on these clusters, and not on the general system call, in order to better capture normality and deviations on a more compact input space. This is important because some system calls, most notably `open`, are used in very different ways. Indeed, `open` is probably the most used system call on UNIX-like systems, since it opens files or devices in the file system creating a descriptor for further use. Only by careful aggregation over its parameters (i.e., the file path, a set of flags indicating the type of operation, and an opening mode) we can de-multiplex the general system call into "sub-groups" that are specific to a single function. In order to do this, we must define a way to measure "distance" among arguments, as we will show.

Afterwards, the system builds models of the parameters inside each cluster. The type of models, as well as the type of distances used for agglomeration, depend on the type of parameter, as shown in Table 5.1. In our framework, the distance among two system calls, s_i and s_j, is the sum of distances between corresponding arguments $D(s_i, s_j) = \sum_{a \in \mathrm{A}_s} d_{\mathrm{model}(a)}(s_i^a, s_j^a)$ (being A_s the shared set of system call arguments). For each couple of corresponding arguments a we compute the distance as:

$$d_a = \begin{cases} K_{(\cdot)} + \alpha_{(\cdot)}\delta_{(\cdot)} & \text{if the elements are different} \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

where $K_{(\cdot)}$ is a fixed quantity which creates a "step" between different elements, while the second term is the real distance between the arguments $\delta_{(\cdot)}$, normalized by a parameter $\alpha_{(\cdot)}$. We use "$(\cdot)$" to denote that such variables are parametric w.r.t. the type of argument.

Since hierarchical clustering does not offer a concept analogous to the "centroid" of partitioning algorithms that can be used for classifying new inputs, we also created, for each cluster, a stochastic model that can be used to classify further inputs. These models generate a *probability density function* that can be used to state the probability with which the input belongs to the model. It is not strictly necessary for such model, or its distance or probability functions, to be the same as the distance functions that are used for clustering purposes.

As can be seen in Table 5.1, at least 4 different types of arguments are passed to system calls: path names and file names, discrete numeric values, arguments passed to programs for execution, users and group identifiers (`UID`s and `GID`s).

Path names and file names are very frequently used in system calls. They are complex structures, rich of useful information, and therefore difficult to model properly. For the clustering phase, we chose to use a very simple model, the directory tree depth. This is easy to compute, and experimentally leads to fairly good results. Thus, in Equation 5.1 we set δ_a to be the difference in depth. The stochastic model for path names is a probabilistic tree which contains all the directories involved with a probability weight for each. Filenames are often too variable to be considered, so if the leaves of the tree are too different we simply ignore them for that specific model.

Discrete numeric values such as flags, opening modes, etc. are usually chosen from a limited set. Therefore we can store all of them along with a discrete probability. Since in this case two values can only be "equal" or "different", we set up a binary distance model for clustering, where the distance between x and y is:

$$d_a = \begin{cases} K_{disc} & \text{if } x \neq y \\ 0 & \text{if } x = y \end{cases}$$

and K_{disc}, as usual, is a configuration parameter. In this case, the generation of the probability density function is straightforward.

We also noticed that *execution arguments* (i.e. the arguments passed to the `execve` syscall) are difficult to model, but we found the length to be an extremely effective indicator of similarity of use. Therefore we set up a binary distance model, where the distance between x and y is:

$$d_a = \begin{cases} K_{arg} & \text{if } |x| \neq |y| \\ 0 & \text{if } |x| = |y| \end{cases}$$

denoting with $|x|$ the length of x and with K_{arg} a configuration parameter. In this way, arguments with the same length are clustered together. For each cluster, we compute the minimum and maximum value of the length of arguments. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if its length belongs to the interval, and 0 otherwise.

We developed an ad-hoc model for *user and group* identifiers. These discrete values have three different meanings: `UID` 0 is reserved to the super-user, low values usually are for system special users, while real users have `UID`s and `GID`s above a threshold (usually 1000). So, we divided the input space in these three groups, and computed the distance for clustering using the following formula:

$$d_a = \begin{cases} K_{uid} & \text{if belonging to different groups} \\ 0 & \text{if belonging to the same group} \end{cases}$$

and K_{uid}, as usual, is a user-defined parameter. Since `UID`s are limited in number, they are preserved for testing, without associating a discrete probability to them. Fusion of models and incorporation of new elements are straightforward. The probability for a new input to belong to the model is 1 if the UID belongs to the learned set, and 0 otherwise.

In order to take into account the execution *context* of each system call, we use a Markov chain (i.e. a first order Markov model) to represent the program flow. The model states represent the system calls, or better they represent the various clusters of each system call, as detected during the clustering process. For instance, if we detected three clusters in the `open` syscall, and two in the `execve` syscall, then the model will have five states: $\texttt{open}_1$, $\texttt{open}_2$, $\texttt{open}_3$, $\texttt{execve}_1$, $\texttt{execve}_2$. Each transition will reflect the probability of passing from one of these groups to another through the program. A sample of such a model is shown in Figure 5.2. This approach was investigated in former literature [20, 21, 37, 63, 44, 48], but never in conjunction with the handling of parameters and with a clustering approach.

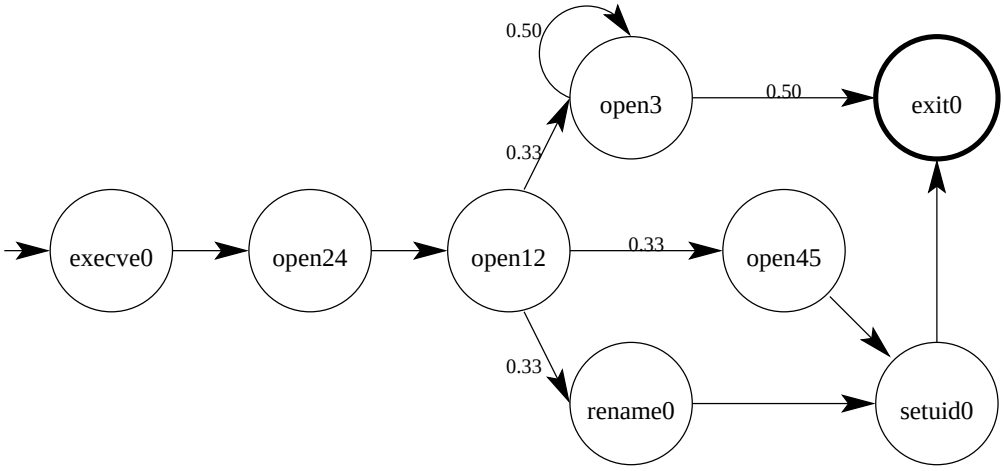

Figure 5.2: A sample of the resulting Markov model with the clusters of system calls as states

During training, each execution of the program in the training set is considered as a sequence of observations. Using the output of the clustering process, each syscall is classified into the correct cluster, by computing the probability value for each model and choosing the cluster whose models give out the maximum composite probability along all known models: $\max(\prod_{i \in \mathrm{M}} P_i)$. The probabilities of the Markov model are then straightforward to compute.

Since training should happen, ideally, on the machine which will be monitored, it is important to notice that the prototype is resistant to the presence of a limited number of outliers (e.g. abruptly terminated executions or attacks) in the training set, because the resulting transition probabilities will drop near zero. For the same reason, it is also resistant to the presence of any cluster of anomalous invocations created by the clustering phase. Therefore, the presence of a minority of attacks in the training set will not adversely affect the learning phase, which in turn does not require an attack-free training set, and thus it can be performed on the deployment machine.

During the detection phase, each system call is considered in the context of the process. The cluster models are once again used to classify each syscall into the correct cluster: the probability value for each model is computed and the stored cluster whose models give out the maximum composite probability ($P_c = \max(\prod_{i \in \mathrm{M}} P_i)$) is chosen as the "system call class". Three distinct probabilities can be taken into account in order to build anomaly thresholds:

- P_s, the probability of the *execution sequence* to fit the Markov model up to now;
- P_c, the probability of the *system call* to belong to the best-matching cluster;
- P_m, the *latest transition* probability in the Markov model.

We fuse the last two into a probability value of the single syscall, $P_p = P_c \cdot P_m$. A second, separate value for the *sequence probability* P_s is kept. Using the training data, appropriate threshold values are calculated by considering the lowest probability over all the dataset for that single program (for both P_s and P_p). We then choose a sensitivity parameter for scaling such value, giving the final *anomaly threshold*. A process is flagged as malicious if either P_s or P_p are lower than the anomaly threshold. For avoiding a P_s which quickly decreases to zero for long sequences, we introduced a "scaling" of the probability calculation based on the geometric mean, by introducing a sort of "forgetting factor": $P_s(l) = \sqrt[2l]{\prod_{i=1}^{l} P_p(i)^i}$ (where l is the sequence length). In this case, we demonstrated [55] that $\mathrm{P}\left[\lim_{l\to+\infty} P_s(l) = 0\right] = 1$, but it converges more slowly. Experimentally, this latter scaling function leads to much better results in terms of false positive rate.

One of the reasons why this type of detection is deemed useful in the context of WOMBAT is that, nowadays, skilled attackers are wary of writing anything on the hard drive of an attacked machine. Thus, if we wish to preserve malware samples, we need to take into account in-memory execution, which is a widely known and used "definitive anti-forensic" [28, 17, 34] technique.

There are two wide classes of anti-forensics techniques: *transient* techniques make the acquired evidence difficult to analyze with a specific tool or procedure, but not impossible to analyze in general. *Definitive* anti-forensics techniques instead effectively deny once and forever any access to the evidence. In this case, the evidence may be destroyed by the attacker, or may simply not exist on the media. The final objective of anti-forensics is to reduce the quantity and spoil the quality [32] of the evidence that can be retrieved.

Examples of transient anti-forensics techniques are the fuzzing and abuse of filesystems in order to create malfunctions or to exploit vulnerabilities of the tools used by the analyst, or the use of log analysis tools vulnerabilities to hide or modify certain information [27, 32]. In other cases, entire filesystems have been hidden inside the metadata of other filesystems [32], but techniques have been developed to cope with such attempts [64]. Other examples are the use of steganography [45], or the modification of file metadata in order to make filetype not discoverable. In these cases the evidence is not completely unrecoverable, but it may escape any quick or superficial examination of the media: a common problem today, where investigators are overwhelmed with cases and usually undertrained, and therefore overly reliant on tools.

Definitive anti-forensics, on the other hand, effectively denies access to the evidence. The attackers may encrypt it, or securely delete it from filesystems (this process is sometimes called "counter-forensics") with varying degrees of success [30, 29]. Access times may be rearranged to alter the activity timeline that is usually exploited by analysts to correlate events. The final anti-forensics methodology is not to leave a trail: for instance, modern attack tools (commercial or open source) such as Metasploit [3], Mosdef or Core IMPACT [24] focus on pivoting and in-memory injection of code: in this case, nothing or almost nothing is written on disk, and therefore information on the attack will be lost as soon as the system is powered down, which is usually standard operating procedure on compromised machines. These techniques are also known as "disk-avoiding" procedures.

Memory dump and analysis operations have been advocated in response to this, and tools are being built to cope with the complex tasks of reliable acquisition [19, 69] and analysis [19, 67, 61] of a modern system's memory. However, even if the memory can be acquired and examined, if the injected process has already terminated, no trace of the attack will be found: these techniques are much more useful against in-memory resident backdoors and rootkits, which by definition are persistent. In order to detect in-memory malicious code, we propose the use of S^2A^2DE.

As a proof of concept, we generated used two attacks on two console applications: `bsdtar` and `eject`, on an Intel x86 machine running FreeBSD 6.2. These two applications have been recently found to be vulnerable to two different buffer overflow vulnerabilities that allow to execute arbitrary code. In the case of `mcweject 0.9`, the vulnerability [58] is a very simple stack overflow, caused by improper bounds checking. By passing a long argument on the command line, an aggressor can execute arbitrary code on the system with root privileges. There is a public exploit for the vulnerability [35] which we modified slightly to suit our purposes and execute our own payload. The attack against `bsdtar` is based on a publicly disclosed vulnerability in the PAX handling functions of `libarchive 2.2.3` and earlier [59], where a function in file `archive_read_support_format_tar.c` does not properly compute the length of a buffer when processing a malformed PAX archive extension header (i.e., it does not check the length of the header as stored in a header field), resulting in a heap overflow which allows code injection through the creation of a malformed PAX archive which is subsequently extracted by an unsuspecting user on the target machine. In this case, we developed our own exploit, as none was available online, probably due to the fact that this is a heap overflow and requires a slightly more sophisticated exploitation vector. In particular, the heap overflow allows to overwrite a pointer to a structure which contains a pointer to a function which is called soon after the overflow. So, our exploit overwrites this pointer, redirecting it to the injected buffer. In the buffer we craft a clone of the structure, which contains a pointer to the shellcode in place of the correct function pointer.

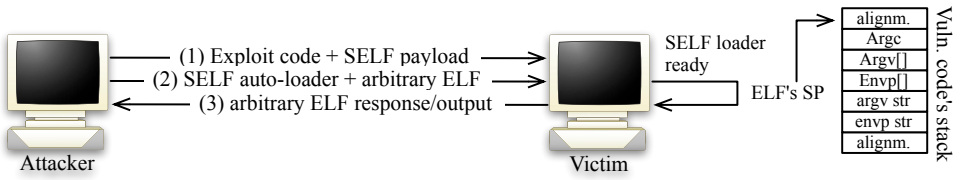

Figure 5.3: An illustration of the in-memory execution technique we developed and used for this work

We also developed a modified version of SELF [8], which we improved in order to reliably run under FreeBSD 6.2 and ported to a form which could be executed through code injection (i.e., to shellcode format). This tool implements a technique known as "Userland Exec": by overwriting the program headers of any statically linked ELF binary, and by building a specially-crafted stack it allows an attacker to load and run that ELF in the memory space of a target process without calling the kernel and, more importantly, without leaving any trace on the hard disk of the attacked machine. This is accomplished through a two-stage attack where a shellcode is injected in the vulnerable program, and then retrieves a modified ELF from a remote machine, and subsequently injects it into the memory space of the running target process, as shown schematically in Figure 5.3. In our proof of concept installation, we obtained excellent results in terms of detection, and further testing is ongoing.

S^2A^2DE is currently implemented in C. Both the clustering phase and the behavioral analysis are multithreaded, and the results of both procedures are stored in a binary format but can be dumped in XML for manual inspection, if needed. At runtime, the prototype dinamically loads the program profiles it needs, and stores them in memory. The prototype can send output to standard output, syslog facilities, and/or to log files in IDMEF format.

We profiled the code with `gprof` and `valgrind` for CPU and memory requirements. The throughput for the training phase varies between 6120 and 10228 syscalls per second. The training phase is also memory consuming, with a worst-case peak during our tests of about 700 MB. The performance observed in the detection phase varies between 12395 and 22266 syscalls/sec. Considering that the kernel of a typical machine running services such as HTTP/FTP on average executes system calls in the order of thousands per second (e.g., around 2000 system calls per second for `wu-ftpd` [57]), the overhead introduced by S^2A^2DE is noticeable but does not severely impact system operations.

5.2 Malware Detection by Attributed-Automata

Results from this section have been submitted to the IEEE Symposium on Security and Privacy (SSP'09).

This deliverable focuses on the possible specification languages for malware behaviors. Therefore, the detection process is not completely described in this document but will be in the dedicated Deliverable "D16 (D4.2) Analysis report of behavioral features (D4.2)". However, since the abstract specification called the Abstract Malicious Behavioral Language (AMBL) from Section 3.2.2 is specifically designed for detection, the global detection architecture is briefly introduce to show how the language is integrated in the overall process.

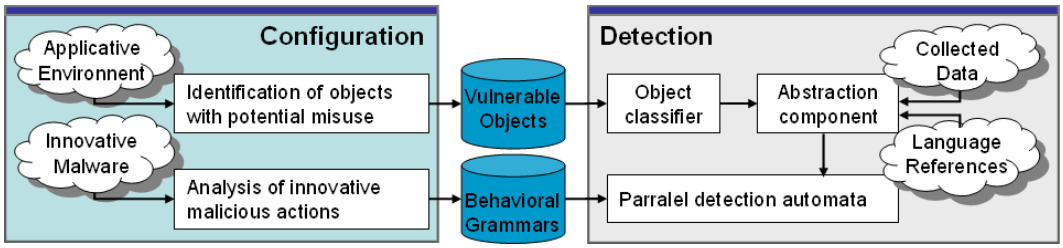

Figure 5.4: Configuration and detection processes.

The main usage of the language is the description of the malicious behavioral signatures which are defined by sub-grammars of this language. The original signature generation is based on a tight analysis of innovative malware i.e. malware introducing new malicious techniques unseen in the numerous variations from known strains. Determining whether the thousand of collected malware every day are new instances or variants of well-known families can be addressed by clustering techniques as described in Section 4.3. Once the new malware identified and their innovative behaviors described, the resulting grammatical behavioral signatures are stored in a dedicated database used to fed parallel parsing automata. When a given automaton reaches an accepting state, the grammatical description has been successfully parsed and the corresponding behavior has been detected. The ongoing process from signature generation to detection is described in the Figure 5.4.

Parsing automata can only process data written in the same representation that the grammars they parse. Translation from the raw collected traces into the abstract language is thus required to feed detection as described once again in the Figure 5.4. The translation process, as well as its initial configuration based the considered platforms

and languages, is wholly described in the Section 3.2.3. The resulting translation mechanism has been integrated in the detection architecture as pictured in Figure 5.5. In this same Figure, it can be observed that the scheduled architecture will support translation from executable trace collectors for PE Executables and script analyzers for Visual Basic Scripts.

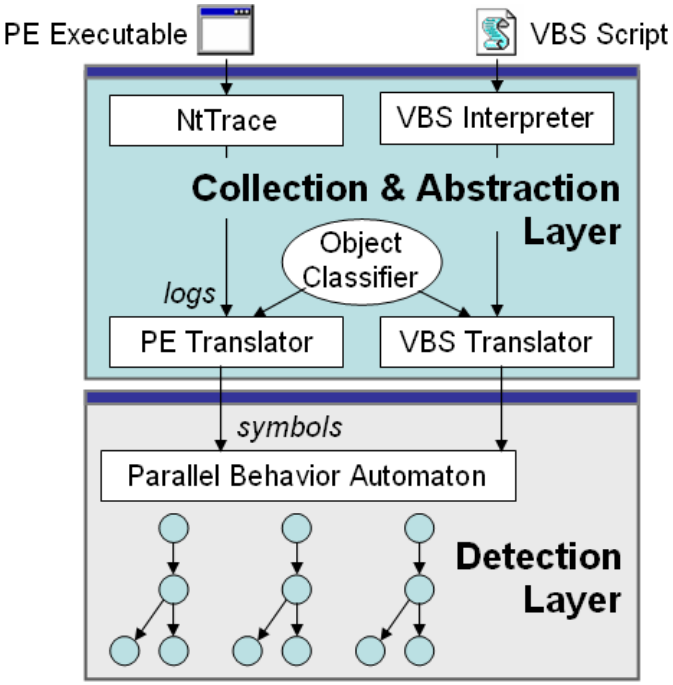

Figure 5.5: Multi-layered detector architecture.

5.3 Malware Slicing for Information Flow-based Detection

Recently, the malware writers' focus has changed from the problem of distribution to that of concealment. Currently, the predominant number of threats circling on computer networks employ some means of binary concealment. Considering this trend, it is surprizing to see that the vast number of intrusion detection system available on the market are still mainly based on matching binary patterns inside a program image. Although a rather intuitive and fast approach, its various shortcomings have been discussed throughout this document.

To battle this deficiency in signature matching, the research community has come up with various attempts to detecting generic malicious behavior or finding known patterns of malware actions inside unkonwn running executables [65, 23, 38]. Instead of focusing on an executable's image, this *behaviour based detection mechanism* is grounded on monitoring a program's actions (i.e. its system call invocations to create/modify files, interact with remote hosts, modify the system environment, etc.) and blacklisting certain (sets of) action sequences.

Unfortunately, previous approaches on this field have suffered from a number of shortcomings. Analogous to binary obfuscation, malware can generate code to invoke additional system calls generating noise or reorder the execution of certain actions to evade detection through call signatures. Furthermore, certain malicious activities, such as the propagation through mass-mailing a malware's own binary image, are very hard to detect on a system call basis. In the particular case of a mass-mailer, a system call monitor would only see reading of a file while maintining a network connection to a remote host. Distinguishing such activities from various benign programs' like email or chat clients is therefore particularily challenging.

Another possibility to attack pure system call based detection mechanisms are so-called mimicry attacks [70]. In such an attack, the malware attempts to immitate a benign program's system call signature to evade detection or create a too high false positive rate that renders the signature unusable.

To address the insufficiency of extracting binary characteristics and monitoring at system call level to allow malware detection, we consider extending the detection schema by two steps in this chapter. In a first step, we try to minimize the impact of random system call noise by generating system call sequence signatures that contain only calls mandatory for the action under observance (e.g. propagation). Then, we verify alerts triggered by the system call signature in a second step.

During this verification step, our system tries to predict what data the allegedly detected malware would have generated, provided the same input as processed by the program under inspection (note that our system has full knowledge of data consumed by a program as it monitors from a kernel-level). If the anticipated output and that obtained through monitoring the binary share salient patterns, a clear indication of the execution of the suspected malware is given and the system can issue appropriate actions.

5.3.1 Our approach

As briefly broached in the introductory text, our detection technique is a two-part process. Firstly, we try to detect the immutable execution characteristics of a malware (family) that our analysis shows to be mandatory for the successful execution of the

malware's actions. In a second step, our scanner tries to predict system call arguments of the candidate binary that are then compared to the actually observed ones to eliminate false positive alerts.

This poses four basic challenges to our approach that we will discuss individually in this section. Firstly, we will discuss how we generate system call signatures, followed by an explanation of how our system monitors running executables to match the previously generated signatures. Next, we will deal with the problem of extracting specific behavior from a binary in order to be able to predict future system calls' arguments. In the last part, we will describe, how these extracted behavior profiles are used to verify alerts raised by the system call signature matching.

System Call Signatures Extraction

To learn about a specific malware's internal actions, we first execute a copy of the malicious program inside an extended version of Anubis [14, 9], the enhanced version of the Qemu [16] full system emulator already mentioned in a previous deliverable. We monitor all accesses to Windows' system interface, tainting memory areas and registers containing data provided by the system kernel. Subsequently, we propagate taint labels using four different heuristics:

- *direct propagation*: Taint destination registers, processor flags, and memory areas touched by instructions using tainted sources,
- *indirect propagation*: Taint values that are read from memory using tainted address values,
- *conditional propagation*: Taint register and memory initializations occurring within a branch that was executed due to a tainted processor flag, and
- *program propagation*: Include each process in the taint analysis that is either started by or consumes data from a process that contains tainted information.

Additionally to the propagation, we also log the current instruction pointer of code that accesses tainted information. At the same time, we extract all instructions executed by the binary, log the observed path/control flow through the binary's image, and store accesses to memory in order to facilitate later enhanced static analysis of the program, even in the case of packed binaries.

Thus, in a later, offline analysis, we can see exactly, which initial system calls provided information (from now referred to as *taint sources*) that has influence on subsequent system call arguments (*taint sinks*). In cases where the binary modifies data before passing

it to the kernel (as in the email propagation example above where such a modification could manifest as a base64 encoding), we can also pinpoint exactly which instructions account for this transformation.

Based on this taint propagation, we can then construct system call signatures. We do this by inspecting all system calls sinking tainted data. For each such call, we can recurse backwards on the associated taint labels to find the corresponding taint sources. Although the mere fact of finding certain system calls in a chronological order does not imply any enforced ordering in general, having a taint relation between the calls does. This reasoning is sane, as the taint relation prescribes that the latter system call consumes data provided by the former.

Often, we can repeat this recursion multiple times, as the taint source's system call typically is also a taint sink for another chain of taint relations. Therefore, we can generate long chains of chronologic system calls that must occur for the initially inspected call to happen. In a last step, we simplify the system call sequences by detecting and eliminating system call cycles.

Again, consider the mass mailing worm example: When monitoring a call to a *send* system call[1] containing part the worm's image, we expect to find taint labels associated with the provided buffers. Typically, these labels will show a connection to one of the *Readfile* or *NtMapViewOfFile* system calls. These calls, in turn, will yield a dependency on a previous *NtOpenFile* or similar, creating a very specific chain of events that must be observable.

Signature Matching

For the purpose of matching the signatures obtained throught the process described above, we wrote a Windows XP kernel driver. Once installed, this driver hooks the system service dispatch table, to be able to monitor all system calls invoked by user-land programs.

The scanner has two primary objectives: Firstly, it keeps a log of all system calls invoked by a program along with the respective call arguments. Secondly, it continously matches sequence signatures on the call logs to find suspicious binaries. Whenever it encouters a matching signature, all available information is passed to a user-land program that which, in turn, tries to verify the alert as described in below sections.

[1] *Send* by itself cannot be monitored as system call in the Windows operating system. Instead, we search for *NtDeviceIoControlFile* with a specific set of flags set.

Malware Slicing

In this chapter, we use the term *malware* or *binary slicing* to refer to the process of extracting only a specific subset of actions from a typically much larger set of activities performed by a (possibly malicious) binary.

Looking at actions usually performed by a virus, such slices could contain any of the following:

- copy the program's binary into a system directory,
- restart execution as different user or as Windows service,
- create (polymorphic / metamorphic) copies of itself,
- send the binary image to remote hosts using email, vulnerable remote services, etc.

For this project, we are particularly interested in actions that modify data provided by the operating system in some way to then use it as (part of) an argument to another system call. Considering the examples provided above, all four match this criteria since either the binary's current execution path, information about the location of system directories, and access to the program image itself are all provided through a system call.

Slice Extraction As a first step of the slicing procedure, we have to select an *interesting* taint relation (i.e. a connection from a taint source to its sink). This is necessary, as some system call sequences propagate kernel identifiers without representing an explicit encoding algorithm (e.g. the use of a common socket handle between connect and send calls).

After selecting a specific sinking taint label, our analysis tool automatically includes all taint labels that sink in the same system call invocation (e.g. tainted *length* attributes for a *send* system call) as well as all other sinking labels that are passed to other invocations of the system call using the same handle (e.g. including all data sent using a common socket handle).

In a next step, the algorithm recurses all selected source-sink chains, finding all initial labels that are required to calculate the sinking labels' values. Simultaneously, it collects all intermediary instructions encoding the value transformation, if such a transformation is observed.

Slice Containment Clearly, only including instructions that touch tainted data is not sufficient to extract an algorithm from the analyzed binary. For one thing, it is very

unlikely that all arguments of a system call have associated taint information. This is especially true for flag parameters that are usually immediate values pushed onto the stack just before the system call invocation or string parameters that are only partly made up from tainted characters.

For another thing, *indirect taint propagation* often requires memory or register content that need neither have a close spatio nor temporal relation with instructions accessing tainted information.

In order to produce self-contained code slices, we implemented a slicing algorithm similar to one described in [73]. Initial tests showed that due to the complexity of most malware samples, it was infeasible to create and analyze complete procedure-dependency graphs. We therefore decided to implement, what Zhang et. Al. describe as *no preprocessing without caching* algorithm.

Our algorithm starts at the latest system call invocation selected during the initial step of the slicing process. For each selected system call, we provide the algorithm with the function signature, so it can insert the adequate number of initial elements into the set of undefined dependencies. It then uses the previously observed control flow to step *backwards* through the binary. Each single instruction is disassembled and in case it fulfills an undefined dependency, it is added to the slice, along with its respective new dependencies. All other instructions are replaced by equally sized NOP instruction sledges to fill the gaps and maintain correctness of relative jumps.

To resolve dependencies that cannot be decided with pure static analysis, we use the memory access logs produced during the emulation. For each such memory-read dependency, the access logs are traversed to find the previous write having the same memory address, tagging the corresponding instruction. As soon as the algorithm traverses previously tagged instructions, it adds them to the slice along with their respective dependencies. All memory accesses that do not have such previous writes (e.g. because they are accesses to statically initialized data segments or BSS sections) are treated specially. We will deal with this situation in more detail in a later section.

Each time, the control flow recurses into a subfunction (regardless, if the call target is a system call or standard function), the algorithm analyzes the subcall and includes it, if one of the following criteria is fulfilled:

- the analysis has an unfulfilled dependency for register *eax*[2] before stepping back into the function,
- the called function (or any subfunction called during its execution) contains tagged instructions to fulfill a later memory read, or

[2] All standard compilers currently use register *eax* to pass the function result to the calling code.

- a later invocation of the same call was included in the slice previously (e.g. when analyzing calls inside a loop).

Otherwise, the call instruction and all associated parameter pushes are replaced by NOP instructions.

As soon as the function detects that it reaches the first instruction of the analyzed function[3], we employ static analysis of the function body to find the number of parameters required by the function. This analysis can be improved throught the information of currently undefined dependencies, having a positive offset to one of the stack registers. Using this information, we can recurse into the caller and restart the analysis process at the position of the subfunction call.

Whenever we detect a function beginning whose caller is not yet part of the slice, we analyze the set of undefined dependencies and search for tagged instructions that are not yet included in the slice. If these sets do not contain any elements and all initially selected calls to taint-sinking system calls have been included, we can stop the slicing process. This is because, at this point, the slice is self-contained[4], embodies all interesting actions, and we can use the function call as starting point to the slice.

Currently, our prototype binary slicer is able to handle machine code generated from standard C and C++ code as well as human written/optimized assembler code. Intuitively, we expected to get small slices since programmers often encapsulate basic actions (e.g. propagation through email, manifest inside the operating system, etc.) in small units or functions. Our experiments quickly proofed this assumption to be correct, allowing our prototype to produce working, self-contained slices in a matter of a few seconds in most cases.

For malware that use very long execution traces to accomplish certain actions, we are also able to successfully create slices in matters of minutes. But we leave it as a matter for future work to implement adequate caching strategies to reduce this overhead (note: Although it is desirable to have the slicing process as performant as possible, we typically have to run the algorithm only once per action and malware family to create a signature. Thus, we consider even larger delays acceptable).

Slice Retention To store a generated slice, the main function (i.e. the function containing the entry point) and all functions called directly or indirectly by it are embedded inside a Microsoft Windows DLL. All internal function calls are patched to match the correct location inside the library. The main function's location is exported and is thus the only callable entry point from outside the slice.

[3] We observe this by a sudden change of instruction address, followed by a *call* instruction.

[4] Here, self-contained denotes that all registers, flags, and memory values will be set before being read.

To facilitate later replaying and sandboxing of the slice code, all calls to system calls are replaced with calls to a special function block. In this block, the slice instead calls a function it looks up in a call table, similar to Windows' system service dispatch table. Prior to invoking the slice, this table must be set up by the environment to contain a function pointer for each invokable system call. This facilitates passing the same data to the slice as previously consumed by the program under inspection. Similarly, all API functions (e.g. calls to malloc) recognized during the slicing process are also replaced with a callback to make the generated code smaller and more stable.

The DLL also contains another externally callable function. This second function can be used to query the slice about required, externally defined memory areas. Such dependencies are typically generated when accessing BSS sections (as mentioned above) and need to be set up correctly before calling the slice's entry point.

In an initial attempt, we considered letting the slice set up all BSS sections using the values observed during the malware emulation. Although this is a feasible approach, various observations made on real malware lead us to not do this for the following reason: Often, viruses and worms use static BSS data as key for various operations like creating files or accessing Windows registry keys. As these values are located at fixed addresses inside the binary image, this provides malware authors with a trivial means to morph the actions of the program, by simply overwriting these values before propagation.

Thus, we decided to compel the environment calling our sliced algorithm to first query the current values of these memory addresses from the binary the slice is supposed to be compared to. The queried values are then set up accordingly before invoking the slice's main routine.

As a positive side effect, this even allows us to use fewer, generic signatures to detect a broad range of polymorphically modified variants of one malicious program.

Slice Replaying and Matching

Whenever the kernel-level system call monitor recognizes a signature inside the call sequence of a running process, a user-land *scanner* is invoked to verify the alert. For this purpose, the kernel informs the scanner, which running process triggered the alarm, what signature was matched, along with the log of recently monitored system calls and their respective arguments. The scanner can then load the slice associated with the given signature and start the replaying process.

In a first step, the scanner loads the DLL into its address space and queries the slice about required, preinitialized data sections. It then connects to the suspicious binary to read out the required data sections and replicate the gathered data inside its own address space. Next, the addresses of the internal callback functions, representing all

invokable system calls, are copied to the call table. Finally, the slice's entry point is invoked inside a separate thread. This allows the scanner to trap crashes and prevent unexpected interactions with the scanner's internal state.

The sliced algorithm will then start querying the scanner for available input using the callback functions. During each such invocation, the scanner uses the system call arguments provided by the kernel-land monitor to provide the slice with the same data as the suspicious program. Whenever the slice invokes a system callback that passes data *to* the scanner (e.g. in calls like *send*, *WriteFile*, or *CreateProcess*), the provided arguments can be matched with data actually observed in the monitored program.

As soon as the generated and observed call arguments share sufficiently common parts, the scanner can confirm the alert and inform the kernel to take adequate actions. If no matches are observed over a certain number of system callbacks or the passed arguments are sufficiently diverging, however, the replaying is terminated. Again, the scanner informs the kernel who in turn continues to monitor the process for further possible signature matches.

A third, rather common situation, causes the sliced algorithm to crash during execution. Typically, this happens when the scanner was unable to set up all preinitialized data sections correctly, because the monitored program did not provide values at the specified addresses or the inspected data did not have the expected type (e.g. the slice expected a null-terminated string, but an integer was provided). This case is caught by a set of signal handling functions inside the scanner, allowing it to terminate the replay procedure securely. As this is also a clear indication for a mismatch between slice and monitored program, the kernel is informed like in the case of mismatching system call arguments.

5.3.2 Evasion Techniques

Malware detection and concealment is an ongoing arms race. Obviously, once our detection mechanism has established itself on major platforms, virus programmers will come up with various techniques to circumvent detection. To get a feeling for the different approaches of concealment, this section gives an overview of this, together with possible solutions to stay one step ahead.

Hindering signature generation

A basic requirement of our system is that we can observe a sample's malicious activities inside our system emulator. Furthermore, we require to find tainted chains between data sources and the corresponding sinks. If a malware accomplishes to circumvent any of

these two required steps, our system can neither generate system call signatures, nor find a starting point for the slicing process.

To tackle the first obstacle, we leverage the fact that our system is based on an unaccelerated version of Qemu. Since this is a system emulator (i.e. not a virtual machine), it implies that certain trivial means of detecting the virtual environment (as described in [66]) are not applicable. Furthermore, the emulation allows us to mimic specific behaviors of a real computer, including CPU features, hardware identification, and so on.

Finally, we have parallel ongoing projects working specifically on improving the stealth of our system emulator. By introspecting system call invocations and data passed from kernel- to user-land programs, we try to elude system fingerprinting usable for Qemu detection.

For dealing with the second challenge, maintaining taint label propagation, we have put a lot of effort into the taint propagation algorithm. As described in Section 5.3.1, we implemented data and control dependent taint propagation and pursue a conservative approach to circumvent the loss of taint information as much as possible. Surely, this will require further work as soon as we observe threats in the wild targeting this area.

Hindering slice generation

Researchers have proposed various means to evade static analysis of machine or assembly code [56]. As our slicing algorithm clearly relies on its aptitude to analyzing the emulated code, this poses another possibility to circumvent our system from generating the required slices.

As described in Section 5.3.1, we do not rely on pure static analysis, however. To guarantee our algorithm being able to correctly analyze a program's control flow, we store the observed execution path during emulation. Furthermore, we can employ memory access logs to resolve arbitrary access structures and have thus ensured def-use chains, required for slicing.

Since control flow and def-use chains constitute the two main problems in static analysis, we are therefore confident to correctly handle almost any type of code.

Attacking the scanner

Our approach relies on running a piece of the malware inside our user-land scanner. To fulfill its tasks (e.g. program inspection as described in Section 5.3.1), this scanner program must run with the highest privileges available on the system.

Clearly, this gives malware writers the possibility to attack the scanner from inside the generated code (e.g. by crashing its analysis and therefore hinder from reporting its findings to the kernel). Another opportunity could be trying to run the malicious activities from inside the scanner.

We argue that neither of these attacks can succeed for the following reasons: Firstly, the slice can only contain instructions that were observed during the initial run of the malware. Thus, any attack on the scanner must have had occurred during the emulation also. As we run the inspected malware on the emulated system directly, the malware can neither detect nor attack the scanner environment and any attempt to do so will not find its way into the generated slice.

Secondly, we can use various techniques, such as multithreading and segmentation, to keep the slice from accessing or altering internal memory structures of the scanner. Lastly, the only way to access data or interact with the operating system is through the callback functions provided by the scanner. Therefore, the system is able to detect and prevent any attempt to perform malicious activities in the context of the scanner.

Mimicry attacks

A malware can imitate the actions observable from kernel level of a well-known benign program. Such *mimicry attacks*, generate system call signatures that trigger very often on the benign program, rendering generic system call signatures unusable.

Since we only use these signatures for a first level of detection, we argue that it is highly unlikely that a malware author can generate code that mimics the behavior of a benign program while still achieving the intended malicious behavior. Verifying all alerts by replaying the generated slices will clearly eliminate the false positives while still detecting malicious programs.

Changing encoding mechanism

Our system's main focus lies on the detection of data input-output relations and the intermediary encoding algorithm. As soon as a malware writer decides to implement a new encoding format, our slices are rendered useless (for the new malware).

However, completely changing the encoding algorithm contained in a program requires a lot of manual work, as this process can hardly be automated, if at all. Comparing the time required for implementing a new encoding mechanism to the proces of fully automated generation of malware slices, we clearly see higher ground for the detection system. Eventually, this gives us a chance to win the arms race between malware concealment and detection.

Conclusion

Malware binary signatures are failing and current approaches to behavioral detection have not yet provided an accurate replacement so far. In this section, we have proposed an extension to the conventional malicious behavioral detection that uses program slicing to extract data encoding algorithms. These algorithms can then be used to anticipate detailed actions, like system calls and their precise arguments.

We leverage an enhanced version of the full system emulator Qemu with taint analysis and record all input transforming instructions. These instructions are then extracted into Microsoft Windows DLLs that comprise the malware's actions. Furthermore, we generate generic system call sequences invoked by a malware during its execution.

By monitoring unknown executables running on a live system, we can match the previously generated call sequences and invoke a second-level scanner as soon as a program matches any of the sequences. This second stage then uses the encoding DLLs to transform input previously consumed by the suspicious program into an expected output. If the expected and effectively monitored arguments of subsequent system call invocations share salient characteristics, we confirmed the suspicion and can take adequate actions.

Bibliography

[1] Anubis. http://anubis.iseclab.org/.

[2] Behavioral Analysis Report XML schema definition. http://anubis.iseclab.org/xml_schema/.

[3] MAFIA: Metasploit anti forensics investigation arsenal. Available online at http://metasploit.com/projects/antiforensics/.

[4] Ntinternals - the undocumented functions microsoft windows nt/2k/xp/2003.

[5] pcap (format). http://imdc.datcat.org/format/1-002W-D=pcap (accessed on 20081203).

[6] Shelia. http://www.cs.vu.nl/~herbertb/misc/shelia/.

[7] VirusTotal. http://www.virustotal.com/.

[8] Advanced antiforensics – SELF. Available online at http://www.phrack.org/issues.html?issue=63&id=11, 2005.

[9] ANUBIS. http://anubis.seclab.tuwien.ac.at, 2008.

[10] CWSandbox. http://www.cwsandbox.org/, 2008.

[11] Norman Sandbox. http://www.norman.com/microsites/nsic/, 2008.

[12] D. Arthur and S. Vassilvitskii. How slow is the k-means method? In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, New York, NY, USA, 2006. ACM.

[13] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, September 2007.

[14] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, April 2006.

[15] U. Bayer, P. Milani Comparetti, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS) (to appear)*, 2009.

[16] F. Bellard. Qemu, a Fast and Portable Dynamic Translator. In *Usenix Annual Technical Conference*, 2005.

[17] H. Berghel. Hiding data, forensics, and anti-forensics. *Commun. ACM*, 50(4):15–20, 2007.

[18] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, 1997.

[19] M. Burdach. In-memory forensics tools. Available online at `http://forensic.seccure.net/`.

[20] J. B. D. Cabrera, L. Lewis, and R. Mehara. Detection and classification of intrusion and faults using sequences of system calls. *ACM SIGMOD Record*, 30(4), 2001.

[21] G. Casas-Garriga, P. Díaz, and J. Balcázar. ISSA: An integrated system for sequence analysis. Technical Report DELIS-TR-0103, Universitat Paderborn, 2005.

[22] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 2002.

[23] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[24] Core Security Technologies. CORE Impact. `http://www.coresecurity.com/?module=ContentMod&action=item&id=32`.

[25] J. Crandall and F. Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 1996. IEEE Computer Society.

[27] J. Foster and V. Liu. Catch me if you can.... In *Blackhat Briefings 2005*, Las Vegas, NV, August 2005.

[28] S. Garfinkel. Anti-Forensics: Techniques, Detection and Countermeasures. In *Proceedings of the 2nd International Conference on i-Warfare and Security (ICIW)*, pages 8–9, 2007.

[29] S. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *Security & Privacy Magazine, IEEE*, 1(1):17–27, 2003.

[30] M. Geiger. Evaluating Commercial Counter-Forensic Tools. In *Proceedings of the 5th Annual Digital Forensic Research Workshop*.

[31] M. Gheorghescu. An Automated Virus Classification System. In *Virus Bulletin conference*, 2005.

[32] Grugq. The art of defiling: defeating forensic analysis. In *Blackhat briefings 2005*, Las Vegas, NV, August 2005.

[33] J. Han and M. Kamber. *Data Mining: concepts and techniques*. Morgan-Kauffman, 2000.

[34] R. Harris. Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. In *Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06)*, volume 3 of *Digital Investigation*, pages 44–49, September 2006.

[35] "harry". Exploit for CVE-2007-1719. Available online at `http://www.milw0rm.com/exploits/3578`.

[36] T. H. Haveliwala, A. Gionis, and P. Indyk. Scalable techniques for clustering the web. In *WebDB (Informal Proceedings)*, pages 129–134, 2000.

[37] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[38] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. pages 151–180, 1998.

[39] T. Holz, C. Willems, K. Rieck, P. Duessel, and P. Laskov. Learning and Classification of Malware Behavior. In *Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 08)*, June 2008.

[40] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc. of 30th STOC*, pages 604–613, 1998.

[41] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attributed-automata using abstraction from the platform and language. In *Submitted to the 29th Symposium on Security and Privacy (SSP09)*, 2009.

[42] G. Jacob, E. Filiol, and H. Debar. Functional polymorphic engines: Formalisation, implementation and use cases. *Journal in Computer Virology*, Published online, coming in the EICAR'08 Special Issue, 2008.

[43] G. Jacob, E. Filiol, and H. Debar. Malwares as interactive machines: A new framework for behavior modelling. *Journal in Computer Virology*, 4(3, Special TCV'07 Issue):235–250, 2008.

[44] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations (CSFW'01)*, pages 206–219, Washington, DC, USA, June 2001. IEEE Computer Society.

[45] N. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *COMPUTER*, 31(2):26–34, 1998.

[46] D. E. Knuth. Semantics of context-free grammars. *Theory of Computing Systems*, 2:127–145, 1968.

[47] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, 2006.

[48] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Softw.*, 14(5):35–42, 1997.

[49] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the European Symposium on Research in Computer Security*, pages 326–343, 2003.

[50] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the 2003 European Symposium on Research in Computer Security*, Gjvik, Norway, October 2003.

[51] T. Lee and J. J. Mody. Behavioral Classification. In *EICAR Conference*, 2006.

[52] C. Leita and M. Dacier. SGNET: a worldwide deployable framework to support the analysis of malware threat models. In *7th European Dependable Computing Conference (EDCC 2008)*, May 2008.

[53] L.Kaufman and P. Rousseeuw. *Finding groups in data: An introduction to cluster analysis*. New York: John Wiley & Sons, 1990.

[54] J. B. Macqueen. Some methods of classification and analysis of multivariate observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[55] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Trans. on Dependable and Secure Computing*, 2008. accepted for publication.

[56] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *ACSAC*, pages 421–430. IEEE Computer Society, 2007.

[57] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[58] National Vulnerability Database. CVE-2007-1719. Available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-1719`.

[59] National Vulnerability Database. CVE-2007-3641. Available online at `http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-3641`.

[60] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, 2005.

[61] J. Nick L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197–210, december 2006.

[62] U. D. of Defense. *"Orange Book" - Trusted Computer System Evaluation Criteria*. Rainbow Series, 1983.

[63] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of Hidden Markov Models to detecting multi-stage network attacks. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, page 334, 2003.

[64] S. Piper, M. Davis, G. Manes, and S. Shenoi. *Detecting Hidden Data in Ext2/Ext3 File Systems*, volume 194 of *IFIP International Federation for Information Processing*, chapter 20, pages 245–256. Springer, Boston, 2006.

[65] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 377–388, New York, NY, USA, 2007. ACM.

[66] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting system emulators. In *ISC*, pages 1–18, 2007.

[67] S. Ring and E. Cole. Volatile Memory Computer Forensics to Detect Kernel Level Compromise. In *Proceedings of the 6th International Conference on Information And Communications Security (ICICS 2004)*, Malaga, Spain, October 2004. Springer.

[68] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Usenix Large Installation System Administration Conference (LISA)*, 1999.

[69] B. Schatz. Bodysnatcher: Towards reliable volatile memory acquisition by software. In *Proceedings of the 7th Annual Digital Forensic Research Workshop (DFRWS '07)*, volume 4 of *Digital Investigation*, pages 126–134, September 2007.

[70] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.

[71] Wireshark: The World's Most Popular Network Protocol Analyser. `http://www.wireshark.org`.

[72] S. Zanero. *Unsupervised Learning Algorithms for Intrusion Detection*. PhD thesis, Politecnico di Milano T.U., Milano, Italy, May 2006.

[73] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 319–329, Washington, DC, USA, 2003. IEEE Computer Society.