SEVENTH FRAMEWORK PROGRAMME

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

WORLDWIDE OBSERVATORY OF
MALICIOUS BEHAVIORS AND ATTACK THREATS

D17 (D4.4) Final Analysis Report of Structural Features

Contract No. FP7-ICT-216026-WOMBAT

Workpackage	WP4 - Data Enrichment and Characterization
Author	-
Version	1.0
Date of delivery	M30
Actual Date of Delivery	20/07/2010
Dissemination level	Public
Responsible	POLIMI
Data included from	POLIMI,IEU,SYMANTEC,TUV,VU,HISPASEC

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 Marc Dacier
2229 Routes des Cretes
06560 Sophia Antipolis
France

e-mail: Marc_Dacier@symantec.com
Phone: +33 4 93 00 82 17

Contents

1 Introduction **8**

2 Static Clustering **10**
2.1 Feature-based Clustering 10
2.1.1 Results 12
2.2 Content-based Static Malware Clustering 14
2.2.1 Measuring Sample Similarity 14
2.2.2 Clustering Samples 16
2.2.3 Limitations 16

3 Deriving the Structure of Shellcode **19**
3.1 Shellcode Analysis 20
3.2 The difficulty in executing shellcode 22

4 Identifying Dormant Functionality in Malware through Hybrid Analysis **24**
4.1 System Goals and Approach 26
4.2 System Overview 28
4.2.1 Dynamic Behavior Identification 29
4.2.2 Extracting Genotype Models 30
4.2.3 Finding dormant functionality 31
4.3 System Details 32
4.3.1 Genotype Models 32
4.3.2 Genotype Model Extraction 33
4.3.3 Genotype Matching 42
4.4 Evaluation 43
4.4.1 Genotype Model Extraction 43
4.4.2 Genotype Model Accuracy 45
4.4.3 Robustness 48
4.4.4 Genotype Matching Results 50
4.4.5 Performance 52
4.4.6 Limitations 53

5 Conclusions **54**

Abstract

This deliverable is a final report on the experimental results obtained by using structural features to characterize executable code. It discusses and evaluates a number of techniques, based on these features, that have been developed in the context of the WOMBAT project, and aim to provide a deeper understanding of malicious code and of the relations between malicious code samples.

1 Introduction

As the data collected by several WOMBAT partners shows, there is an exponential explosion of the number of newly available malicious software (*malware*) per day. For instance, the amount of samples submitted to VirusTotal [4], an online service to analyze suspicious files with more than 40 antivirus engines, is nowadays in the order of about three million samples per month. Several antivirus vendors also report a growing number of samples that need to be examined on a daily basis in order to keep up with the latest attacks [25, 37].

Such a large number of samples can be partially explained by the easiness with which malware writers can generate new code forks by personalizing existing code bases, or by re-packing existing binaries using code obfuscation tools [21, 27, 39], or the use of polymorphic techniques [6]. As a result, one malware family can be responsible for tens of thousands of samples per month.

For this reason, the development of novel techniques to quickly assess whether or not a given sample is a minor variant of a well-known malware family or a novel sample is an important research result, to which WOMBAT partners dedicated some effort. We needed low-cost techniques allowing us to easily cope with the large number of malware instances produced by polymorphic techniques and to easily distinguish them from new malware variants. Only by discerning these two cases will the security analyst be able to prioritize the usage of more costly analysis tools on potentially new code bases. Within the WOMBAT project, we developed novel techniques to take advantage of information about the structure of a given malware sample. This information can be used in many different ways, for example to cluster a given set of samples based on structural similarity, or to generate a behavioral fingerprint that can then be statically matched against the structure of a set of malware samples. These techniques can then for example be used to quickly determine whether a given sample needs to be analyzed by costly dynamic analysis, or whether static analysis techniques can already extract enough useful information about the sample.

The purpose of this document is to present the results we obtained by analyzing the *structural features* introduced in Deliverable 11 to assess the novelty of a given sample and obtain a preliminary analysis of it. The document is structured as follows: Chapter 2 is split into two sections which discuss different approaches for clustering malware samples based on static, structural features of a given malware sample. Section 2.1 in-

troduces a technique that is solely based on static characteristics that can be inferred by the analysis of the Portable Executable (PE) format headers of Windows executables. The intuition is that headers of executable files are typically not modified by different samples of the same (polymorphic) malware family since even polymorphic packers do not perform any kind of relinking and thus certain static features stay constant. Section 2.2 introduces a complementary technique, namely content-based static malware clustering. We then compute the pairwise difference between all samples and use this information as a metric to cluster malware samples into families.

Chapter 3 describes how Argos, a containment environment for worms and manual system compromises, was extended for structural analysis of the shellcode collected during an attack. This extension is able to analyze shellcode and extract the different layers of unpacking that it employs and the final "real" shellcode that performs useful actions for the attacker.

Chapter 4 reports on the results of a novel approach to combine static and dynamic analysis to enhance the analysis process. The basic insight is that the results of dynamic analysis can also be leveraged to enhance static analysis: based on the results of dynamic analysis we can extract specific control flow information to fingerprint the code itself. This fingerprint can then also be recognized in different, but related malicious code samples. The research brought good results and led to publication in the IEEE Symposium on Security and Privacy.

Finally, Chapter 5 draws our conclusions on the works presented in the Deliverable.

2 Static Clustering

This Chapter reports on two different approaches for clustering malware samples based on static features.

2.1 Feature-based Clustering

In Deliverable D11 we introduced a simple, but fast algorithm allowing to group together samples likely to be polymorphic instances of the same malware variant. The technique clusters malware samples by looking at a set of static characteristics that can be inferred by the analysis of the file characteristics and the analysis of the PE headers of Windows executables.

The intuition underlying this clustering algorithm derives from our experience in looking at the samples collected by the SGNET data set and at their features. We have seen that all the polymorphic techniques observed in the SGNET malware repository tend to randomize a limited amount of *features* of an attack event. The malware writer has to face the tradeoff between the desire of making the detection of the sample as difficult as possible and the practical cost of randomizing its different features. For instance, the polymorphic packer used by the Allaple worm [13] obfuscates and randomizes the data and code sections of each malware instance, but does not perform more expensive operations such as relinking. While the binary content changes at every propagation attempt, the headers of the executable files are not modified and have some immutable characteristics.

Deliverable D11 introduced therefore the idea of "feature-based" clustering, consisting in the automated discovery of invariants over multiple propagations of a specific malware sample. While we are applying the technique specifically to the static characteristics of a malware sample, such methodology is sufficiently generic to be applied to different data types. We will show in Deliverable D18 how we have been able to apply the very same methodology to the propagation context information in order to classify the different propagation vectors.

As described in Deliverable D11 and exemplified in Figure 2.1, the feature-based clustering algorithm starts from information on each propagation event (in this case, focusing on the malware static features) and on the frequency and localization of the

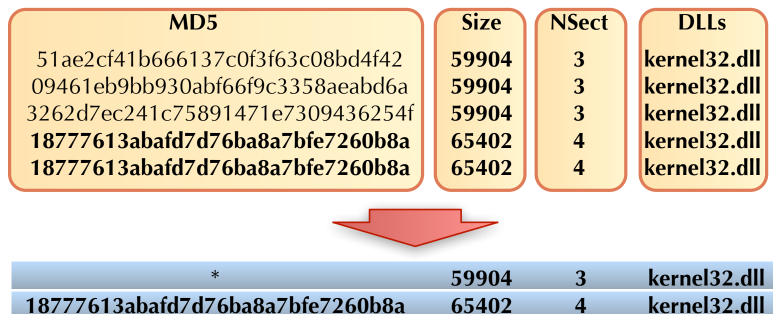

Figure 2.1: Pattern discovery starting from a limited number of invariants (in bold)

Feature	**# invariants**
File MD5	57
File size in bytes	95
File type according to libmagic signatures	7
(PE header) Machine type	1
(PE header) Number of sections	8
(PE header) Number of imported DLLs	7
(PE header) OS version	1
(PE header) Linker version	7
(PE header) Names of the sections	43
(PE header) Imported DLLs	11
(PE header) List of referenced Kernel32.dll symbols	15

Table 2.1: Selected features

event in the dataset (how many times a specific malware sample was downloaded, and the number of targets and attackers involved in its propagation). Then, the methodology discovers frequent patterns through 4 separate phases.

1. **Feature definition.** We define a set of features for each item that we believe to be of interest, and likely to be difficult to randomize by a polymorphic engine. For instance, we consider the content of several PE header fields as reported by PEFile [8].

2. **Invariant discovery.** Each of these features may or may not be randomized by a polymorphic engine. For instance, a non-polymorphic sample will maintain the same MD5 hash over multiple propagation attempts while a polymorphic sample will change its content at every propagation. We need to discover *invariants*, i.e.

non-randomized feature values that can be useful in recognizing a specific malware family. In practice, we consider a feature as invariant if a feature does not change its value across different attack instances (i.e., it is witnessed in multiple code injection attacks in the dataset), different attacking sources (i.e., the same value is used by multiple attackers in different attack instances) and different targets (i.e., multiple honeypot IPs witness the same value in different instances). An invariant value is therefore a "good" value that can be used to characterize a certain event type. Invariant values are represented in bold in Figure 2.1.

3. **Pattern discovery.** Once the invariant values have been identified for each feature, we need to discover the patterns generated by their combination in their dataset. The patterns are discovered by listing all the possible combinations of invariant values, obtained by replacing any non-invariant value with a don't care ('*') field. For instance, Figure 2.1 leads to the identification of two patterns: $\{*, 59904, 3, kernel32.dll\}$ and $\{18777613...b8a, 65402, 4, kernel32.dll\}$.

4. **Pattern-based classification.** Finally, all the identified patterns are applied as classifiers to the original event. If multiple patterns match the very same event (i.e., malware sample), the most specific one is used.

2.1.1 Results

As originally reported in Deliverable D11, we have applied the previously described algorithm to all the malware samples collected by the SGNET deployment. The pattern generation was carried out over 11 different features, listed in Table 2.1. We have been able to corroborate and update the early results that were already provided in Deliverable 11 by classifying 7521 distinct malware samples collected by the deployment and that proved to be valid executable files when executed in Anubis.

First and foremost, the approach proved indeed to be practical and led to the identification of a total of 267 distinct malware classes. The high compression ratio with respect to the original number of malware samples proves that the method is able to cope with the randomization introduced by polymorphic engines in an effective way.

Table 2.2 offers a breakdown of the relationships between malware samples and malware classes according to the number of invariant values associated to each pattern. Out of 11 features, we are able to generate patterns with 9 or more invariants in 93% of the cases: this is a clear indicator of the low level of sophistication of nowadays polymorphic engines.

Since the file hash is one of the features, we are able to clearly identify a total of 60 classes and 60 samples whose hash does not mutate over multiple propagations. These

# of invariants	# of clusters	# of samples
11	60	60
10	42	833
9	129	6072 (81% accumulated samples)
8	24	526
7	7	20
6	3	6
5	1	1

Table 2.2: Degrees of freedom in the clusters

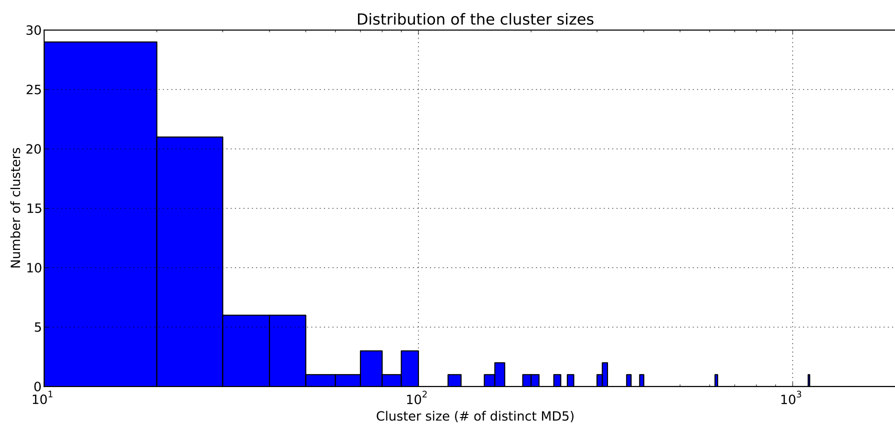

Figure 2.2: Histogram representing the distribution of the size of each identified class in terms of number of distinct samples grouped into it.

samples are therefore non-polymorphic. For the polymorphic samples identified by our method, Figure 2.2 graphically represents the size of the malware classes identified by feature-based clustering in terms of number of distinct samples grouped in each class. Most samples are grouped into few, large clusters likely to be associated to very popular and highly visible polymorphic worms, such as Allaple. The 15 biggest malware classes group a total of 4911 samples, corresponding to 65% of the total. This result underlines the importance of utilizing similar methodologies when analyzing large collections of malware samples: they indeed reduce the task of analyzing thousands of samples to the analysis of a few representatives of each identified malware class, canceling the bias introduced by polymorphic techniques.

2.2 Content-based Static Malware Clustering

A single approach to the malware clustering problem is not always enough. The method discussed in the previous section is effective in the case of polymorphic families, where the content of the PE sections changes dramatically from sample to sample but the PE structure remains similar. However, it is not straightforward to define relationships between the different malware groups. The static clusters generated by the feature-based clustering can be associated to simple recompilations of the same code base, such as different personalizations of the Spybot source code in order to support different Command and Control servers. They can equivalently be representing completely different malware variants, with different features and characteristics. It is difficult, or maybe impossible, to infer from simple features of the malware sample the type of relationship among the different clusters. We have therefore considered a complementary technique, able to fully inspect the malware content in order to infer the amount of overlap among different samples.

2.2.1 Measuring Sample Similarity

The first thing we need to do before we can implement an algorithm for grouping similar samples is to define a way to measure the degree of similarity between two of them. From now on, we will refer to this degree of similarity as the distance between the two samples. The more similar the samples, the smaller the distance between them.

Our problem here is finding a good way of calculating this distance. A common approach to define a distance between two arbitrary sequence of bytes is using the length of their *Longest Common Sub-sequence* (LCS). But the existing algorithms to calculate the LCS are $O(n^2)$, and therefore, slow for practical purposes. Note that this would be the cost only for the comparison, not for the entire clustering (see Section 2.2.3).

This is when delta algorithms come into play. Delta algorithms are aimed to receive two files and generate a set of instructions that can be used to convert one of the files into the other. In theory, delta algorithms are strongly related to finding the LCS of the two files, because finding the optimum set of instruction to convert a file into the other requires solving the LCS problem. In practice however, most delta algorithms sacrifice the optimality of the solution in favor of execution speed.

Algorithm 1 `delta(F1,F2)`

```
N ← 0
i ← 0
while F_1[i] = F_2[i] do
  if i = s_2 then
    return 0
  i ← i + 1
for i ≤ j < i + 65535 do
  x ← four bytes of F_1 at offset j
  T[H(x)] ← j
  j ← j + 1
repeat
  N ← N + 1
  x ← four bytes of F_2 at offset i
  k ← T[H(x)]
  if k is null then
    x ← four bytes of F_1 at offset j
    T[H(x)] ← j
    j ← j + 1
    i ← i + 1
  else
    while F_1[k] = F_2[i] do
      x ← four bytes of F_1 at offset j
      T[H(x)] ← j
      j ← j + 1
      i ← i + 1
      k ← k + 1
until i = s_2
return N
```

To calculate the distance between samples F_1 and F_2 we use algorithm `delta` 1 which

is inspired by xdelta [22]. This algorithm computes the number of instructions necessary to convert one file into the other. In the algorithm the expression $F_x[n]$ references the byte of F_x at offset n, $H(x)$ is a hash function, T is a hash table, and N is the number of instructions. The distance follows the expression:

$$d(F_1, F_2) = \frac{|s_1 - s_2| + 2N}{s_1 + s_2} \tag{2.1}$$

Here s_1 and s_2 are the sizes of the samples F_1 and F_2 being compared. It must be said that this distance is not symmetrical (i.e. $d(F_1, F_2) \neq d(F_2, F_1)$), although it tends towards symmetry when the samples are closer in size. For samples with very dissimilar sizes, but which are still similar in some way (consider, for example, the case of comparing a sample with a truncated version of itself), this asymmetry could lead to very different results depending on the order of comparison. To avoid this discrepancy, we decided that the biggest sample would always be the reference file F_1, and the smallest one the target file F_2. This is because the delta algorithm generates fewer instructions when trying to convert a large file into a smaller one than the opposite way.

2.2.2 Clustering Samples

In order to group the similar samples together a single-linkage clustering algorithm [15, 16] was employed. This algorithm receives a matrix with the distances between the samples of our dataset, and creates a dendrogram showing the similarity relation between them.

When applying the algorithm to a subset of 3950 samples taken from the set of 7521 analyzed in the previous section, 39 clusters were generated containing 738 samples as shown in Table 2.3. As can be seen, the biggest cluster accounts for more than 10% of the samples analyzed, indicating the presence of a big group of samples of the same family within the set.

2.2.3 Limitations

The big disadvantage of this clustering algorithm is its performance. The number of sample comparisons that must be done to fill the initial distance matrix for the clustering algorithm is $\frac{N(N-1)}{2}$. As the number of samples grows linearly, the number of comparisons grows quadratically. The sample comparison algorithm is not very costly in itself, it only depends linearly on the size of the files being compared, but it requires the content of both files to be read completely, incurring a large number of I/O disk operations. With modern hardware, applying this algorithm to up to 5000 samples is

Cluster	# samples
1	423
2	107
3	48
4	18
5	11
6	11
7	9
8	8
9	8
10	6
11	6
12	6
13	6
14	6
15	6
16	5
17	4
18	4
19	4
20	3
21	3
22	2
23	2
24	2
25	2
26	2
27	2
28	2
29	2
30	2
31	2
32	2
33	2
34	2
35	2
36	2
37	2
38	2
39	2

Table 2.3: Clusters generated by content-based static clustering

still affordable, but above this number the time required to complete the process make it unpractical.

3 Deriving the Structure of Shellcode

The Argos emulator [30] is used by various projects to detect zero-day attacks. In Deliverable D11, we motivated an extension of Argos to keep executing shellcode after an attack is detected with an eye on analysing the shellcode. In this chapter we discuss the implementation.

Our aim is to determine the structure of the shellcode. It is arguably impossible to find this structure (or the malware's behaviour for that matter) without executing part of the shellcode. A simple analysis of the instructions in memory reveals very little as the shellcode is typically obfuscated by means of several layers of packing (see Figure 3.1 for a common shellcode structure, and Figure 3.2 for several of these layers in a real exploit). Static analysis may help, but runs into difficulties in the presence of indirect jumps in the code, where the target of a branch cannot be determined statically. Without running the unpacking routines, the bytes in memory are mostly meaningless.

Recall that Argos maintains a separate shadow memory that is not accessible to the guest operating system or its applications. Argos uses this memory to store taint tags. Briefly, it tags all data from a suspect source as tainted. Whenever such data is copied, or used in a calculation, the destination address or register is also tainted. An alert is raised whenever the processor is about to execute tainted instructions or jump to a tainted address.

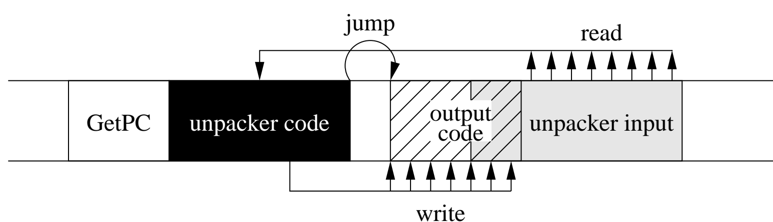

Figure 3.1: Common shellcode: a GetPC routine obtains the current program counter, which is then used by an unpacker that decodes the real shellcode in memory

3.1 Shellcode Analysis

Unpackers work by reading bytes in memory, applying a decoding function and writing the results back to memory. Typically, both the source and the target of the decoding function are in tainted memory areas, although we do not preclude the use of clean bytes. For example, a very simple unpacker is one which simply XORs a range of bytes with a constant. An example is shown in green in Figure 3.2.

After unpacking, the shellcode jumps to this memory area and starts executing the bytes that it previously unpacked here as executable instructions (the `jmp` instruction at address 0x52f830). These instructions may not represent the final shellcode, as attacks often use various layers of packing and unpacking. The final shellcode only reveals itself by doing something useful for the attacker, such as downloading the malware binary and executing it. Doing so from a user space application requires system calls.

In Deliverable D11, we proposed that by continuing the execution of the shellcode and by monitoring jumps to bytes that were written by the shellcode itself, we can find the *structure* of the shellcode, that is, the different layers of unpacking that it employs and the final 'real' shellcode. In a nutshell, we obtain the entry points of each consecutive layer of execution by tracking the shellcode while it executes, including all the bytes that it uses to unpack the shellcode, and then marking all jumps into the bytes that were unpacked at layer n as a transition from to layer $n + 1$.

In this deliverable, we describe how we implemented shellcode structure identification. In addition, we briefly discuss some of the issues we had when implementing it. Finally, we show an example of the resulting ouput of an analysis.

In brief, if an attack is detected, the new version of Argos logs the event and starts shellcode tracking mode. Shellcode tracking mode differs from normal taint tracking mode used by Argos. It is more expensive, but as attacks are rare, we do not consider this to be a problem. In shellcode tracking mode, the actual shellcode instructions are executed until the code reaches a *stop condition*. Typically, the stop condition will be a call to a specific function of the Windows API.

None of this is trivial. It requires obtaining and scanning information structures in Windows to see DLLs are loaded and what the names are of the functions that the function calls. So every call to the Windows API should be monitored. However, we do not want to monitor too much. For instance, we do not track the calls made by the Windows functions internally.

Finally, we are interested not just in the final shellcode, but also in the bytes in the network that correspond to the executed instructions. For this reason, we explicitly log all bytes that are accessed by the shellcode. As mentioned earlier, bytes that are first written by shellcode instructions and subsequently jumped to, indicate a layer transition:

Figure 3.2: Results of Argos' shellcode analysis: the red instructions represent a nop sled, the green code is an unpacker, and the blue instructions are (part of the) real shell code

either the next unpacker, or the real shellcode.

For all of the bytes read and written by shellcode instructions, we track the origins in the network trace. For instance, suppose a byte at address $A1$ in memory originated in the network trace at offset x. If the shellcode reads this byte, decodes it (e.g., by applying an XOR) and writes it back at address $A2$, we still know that the byte at $A2$ derives from offset x in the network trace. If the shellcode subsequently jumps to this address, we know the starting point of the next level in memory, as well as the corresponding network bytes.

The final shellcode is typically reached when the shelcode starts calling its first system call. We now keep executing the shellcode in single step mode, or dump the shellcode to file for later analysis, until we reach a stop condition.

3.2 The difficulty in executing shellcode

Contrary to expectation, executing shellcode in Argos proved to be very difficult. Logging the bytes accessed by shellcode was straightforward and, indeed, for handcrafted attacks we could apply our shellcode analysis fairly easily. However, when applying our solution to real attacks, we noticed that most of them failed to execute and many of them crashed the guest machine. Strangely, some of the exploit/shellcode combinations worked.

After much debugging, we found that the problem is caused by the way in which shellcode figures out the current execution address, i.e., the current value of the program counter. Such code, commonly known as GetPC code is crucial for shellcode that wants to refer to itself (e.g., when packing and unpacking). Several techniques are popular. Perhaps the simplest way to implement GetPC code on x86 is by means of the call instruction[1]

```
$+0:   E8 00000000 CALL $+5 ;      PUSH $+5 onto the stack
$+5:   59          POP ECX  ;      ECX = $+5
$+6: ...shellcode...
```

Assume the current value of the program counter is PC. In this case, the attacker calls the instruction at PC+5 which is exactly the pop instruction. The call instruction will push the return address on the stack, which in this case is also the address of the pop instruction. The pop instruction stores the address in the ECX register and is done.

The disadvantage of the above method is that it contains null bytes, which are awkward – for instance, they do not work well with vulnerable string copies. There are many ways

[1] http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC

around this. As an example, attackers can make the target of the call an address that is lower than the current address. In that case, the offset will be negative, removing the zero bytes. This is also the method that is applied in the green bytes in Figure 3.2. The first green instruction jumps ahead by 0x1b bytes and arrives at address 0x52f832, which immediately performs a function call with a negative relative target address – landing at 0x52f819. The result is that the address that was pushed on the stack (0x52f837) is immediately popped into ESI and the getPC is done.

Still, many attackers prefer to abuse the way in which the x86 deals with floating point instructions to achieve the same effect. The trick is based on a pecularity in the way the x86 deals with floating point operations: the state of the floating point instruction is always kept and can be explicitly recovered using the `fstenv` instruction. The state includes the program counter.

```
$+0 D9EE         FLDZ                ; Floating point stores $+0 in its environment
$+2 D974 E4 F4   FSTENV SS:[ESP-0xC] ; Save environment at ESP-0xC; now [ESP] = $+0
$+6 59           POP ECX             ; ECX = $+0
```

Argos was able to execute shellcode that used the first method of getting the program counter, but not the second. Eventually, we found that the problem was not caused by our code at all. Rather, the Qemu emulator on which we have based Argos, does not always properly handle floating point operations. In certain situations, Qemu does not save the program counter of the floating point instruction.

We fixed the problem by emulating the appropriate actions for all possible FPU instructions in Argos. Whenever an FPU instruction is executed, we explicitly save the program counter in a state field specifically created for this purpose. Doing so is not completely trivial as Qemu translates all x86 instructions in an intermediate code, so it becomes harder to find the program counter of the original instruction. Eventually, we were able to instrument every FPU instruction in Qemu with an explicit call that finds the current program counter.

Figure 3.2 is the output of our new version of Argos in shellcode analysis mode. It shows that the alert is a little friendlier than the existing Argos reports (which we still generate) in the sense that we go beyond simply dumping tainted bytes in memory. Instead, we disassemble the instructions and colour code various unpacking and execution stages of the attackers' code.

4 Identifying Dormant Functionality in Malware through Hybrid Analysis

In Deliverable D11 (D4.3) "Intermediate Analysis Report of Structural Features", we discussed the very early stages of our research on hybrid approaches to malware analysis, based on combining static and dynamic analysis techniques.

Currently, dynamic analysis tools (such as Norman Sandbox, Anubis [1], and CWSandbox [2]) are the most popular choice when performing automated malware analysis. These tools run the binary under inspection in a controlled environment (a sandbox) and monitor its runtime behavior, typically by recording the Windows API library and the operating system calls, including arguments, that the program invokes. The advantage of dynamic analysis techniques is that the actions of a malware sample can be observed directly, without complications due to runtime packing or code obfuscation.

Although useful in practice, dynamic techniques are not without limitations. The most significant issue is that a dynamic analysis run is unlikely to reveal the entire range of capabilities of a given binary. The reason is that the analysis can only observe behaviors for which the corresponding code is actually executed. In contrast, many malware programs include triggers that ensure that certain functions are invoked only when particular environmental or temporal conditions are satisfied. Common examples are bot programs that wait for external input from their command and control servers, or malware programs that execute their malicious payload only before (or after) a certain date.

Previous research [7, 26, 38] has recognized the problem that dynamic techniques suffer from limited coverage. The proposed solutions mainly revolve around the idea of increasing the number of paths that are dynamically explored. To this end, analysis systems execute a binary multiple times. For each run, such systems either provide different inputs that invert the outcomes of certain conditional branches (possible triggers) [7, 26], or simply force the execution along a different path [38]. In both cases, additional code can be reached, potentially revealing previously-unseen behavior.

Unfortunately, systems that explore multiple execution paths have to deal with the *path explosion* problem. Path explosion occurs because, for each interesting branch in the program, the analysis has to follow two successor paths. This leads to an exponential growth in the overall number of paths that need to be explored. Various heuristics are

used to first select more promising continuations. However, these heuristics rarely achieve full code coverage. Thus, even though multi-path analysis can increase the number of behaviors that are observed during a dynamic analysis run, it is unlikely that the entire code is executed. Moreover, multi-path analysis is costly, which is a significant limitation when considering the tens of thousands of samples that need to be analyzed daily.

This research has since found a specific focus in the detection of dormant behavior in malware samples, and has reached a successful conclusion. The results have been presented at the 2010 IEEE Symposium on Security and Privacy, and are published in the conference proceedings.

We proposed REANIMATOR, a novel approach to identify dormant behaviors (behaviors that are not observed during dynamic analysis) in malware binaries. Our approach exploits the fact that many malware samples share the same code base, or at least, parts of their code. This is due to the fact that many samples are just re-packaged, polymorphic variants of the same malware program. Moreover, as previous studies have shown [20], copying and pasting is a common practice in software development, and, certainly, malware programmers are no exception.

The basic approach of REANIMATOR is the following: for every malware sample that is examined by a dynamic malware analysis system, we check its runtime actions for the presence of certain interesting, high-level behaviors. These behaviors are expressed in the context of system calls and Windows API functions, and they represent actions such as packet sniffing, or terminating anti-virus processes. For each behavior that is observed, we automatically locate the code of the binary that is responsible for this behavior. It is important that the located code is accurate; that is, the identified code should be directly responsible for the observed behavior, and not contain unrelated helper functions, such as library routines. Based on the identified code regions, we create a model that captures structural information of this code. Using these models, we can then check other binaries for the presence of similar code. This is done by statically examining the unpacked body of a malware binary. When a model matches, we assume that the malware program contains functionality that implements the corresponding behavior.

The main contributions of the published paper are the following:

- We introduced a novel technique to automatically identify and model code regions in binaries that are directly responsible for specific runtime behaviors.
- We presented a system that leverages models to statically check unknown programs for the presence of previously-seen, malicious functionality.
- Our experimental evaluation demonstrated that our system successfully finds dormant behaviors in malware samples that are not discovered by a dynamic malware

analysis tool.

In the following, we report the main results of the paper.

4.1 System Goals and Approach

The goal of REANIMATOR is to improve the quality of the results delivered by automated malware analysis systems. In particular, we address a key limitation of dynamic malware analysis platforms, which can only examine code paths that are actually executed. To do this, we statically search a malware binary for code that was not run during dynamic analysis but that implements specific functionality that is of interest to a malware analyst. Clearly, the concept of statically searching a program for code fragments that indicate malicious behaviors is not novel *per se*. However, our approach offers a combination of two salient properties that improve significantly over previous work. More precisely, our techniques enable us to *automatically* generate *functionality-aware* models of binary code.

Automated model generation. The ability to *automatically* extract models is important, because it removes the need for tedious and time consuming manual analysis, and scales to the large volume of malware samples that are discovered on a daily basis. Previous work on automated signature (or, more generally, *model*) generation resulted in a number of systems that extract byte strings [18], token sequences [28], or control flow graphs [19] to identify malware binaries. Fundamentally, all these systems share the same underlying mechanism: They search for bytes, instructions, or subgraphs that frequently appear in a set of malicious programs (or execution traces) while, at the same time, they do not appear in legitimate programs (or traces). This basic approach is often successful in automatically extracting models that are able to (statically) classify an unknown program as malicious or benign. However, such models carry little additional semantic information. In particular, it is typically unclear whether a generated model captures some core malware behavior or simply represents a program artifact or auxiliary functionality. This is a serious limitation when such models are deployed in an automated malware analysis system. The reason is that it is often clear that a program under examination is malicious (e.g., because it was collected by a honeypot as the payload of an exploit), but it is not clear which set of functionalities this program implements.

Functionality-aware models. To identify specific malware behaviors, one requires models that are *functionality-aware*. That is, these models need to be equipped with semantic information that indicates the presence of specific, malicious functionality (e.g.,

the fact that a malware sends spam, monitors keystrokes, or starts a web server to provide backdoor access to a compromised host). So far, efforts to build functionality-aware models relied on human analysts. For instance, previous work has proposed semantics-aware code templates [11] and malware blueprints [9]. Such models can precisely characterize code snippets that implement suspicious functionality, such as unpacking or sending spam mails. However, while code templates and malware blueprints are robust to minor code changes and obfuscation, they are nonetheless specific to one concrete way in which a high-level behavior is implemented. We call a piece of code that implements a malicious behavior in one specific way an *instantiation* of this behavior. Of course, it is common that members of a certain malware family share the same instantiation of a particular behavior. Moreover, code sharing makes it likely that the same instantiation can be found across several different malware families. However, it is necessary to manually develop a different code template or blueprint for each new behavior instantiation that is identified. Clearly, this is undesirable, given the massive volume of novel malware that is encountered in the wild.

The ability of REANIMATOR to generate functionality-aware models enables us to statically explore malware binaries for instantiations of specific behaviors. This allows us to accurately recognize malicious program capabilities, even when the corresponding code was not executed, addressing an important limitation of dynamic analysis systems. The ability to extract models automatically allows us to cope with the large number of malware samples that need to be analyzed. In addition, it is faster for our system to automatically generate a model for a newly-identified instantiation of a behavior than for a malware author to manually modify the code to create this instantiation. This is an important advantage in the arms race between defenders and malware authors.

Rationale of approach. As mentioned previously, the goal of our system is to recognize the purpose of code that is *not* executed during dynamic analysis (dormant functionality). To this end, we exploit the fact that a dynamic malware analysis platform receives, executes, and observes thousands of malware programs every day. The basic insight is that, when analyzing a malware sample, we can take advantage of the wealth of information obtained from previous analysis runs. More precisely, we can statically search a program for the presence of a code fragment that is sufficiently similar to code that *(i)* was executed during a previous, unrelated analysis run and *(ii)* was found to be an instantiation of a certain malware behavior. In this case, we know that the program under examination contains functionality that can produce this observed behavior.

It is important to note that the behaviors that can be observed in a dynamic analysis environment vary significantly, even for instances of the same malware family. For example, depending on the availability of the *command and control* (C&C) server or the

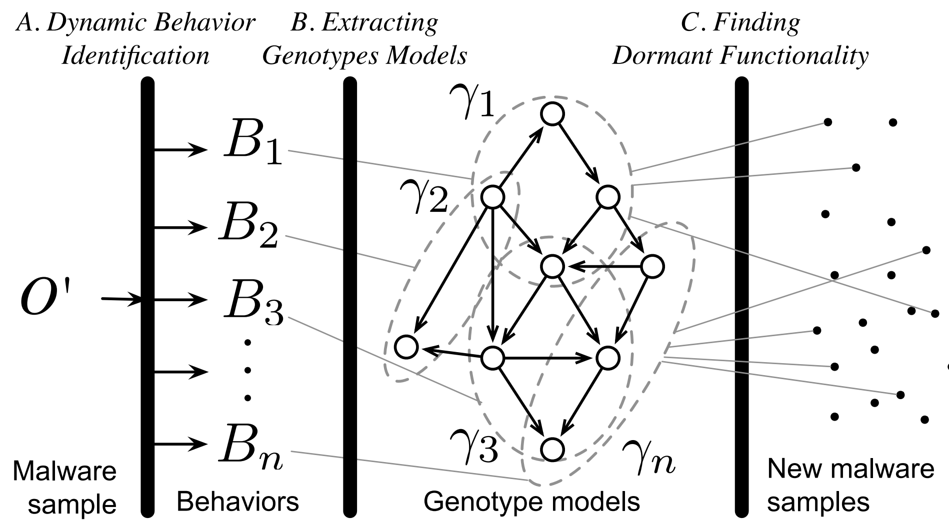

Figure 4.1: An overview of the Reanimator workflow.

currently-advertised command, a bot will invoke different payload routines (e.g., the bot might scan, start a proxy server, send spam, or do nothing). Thus, frequently, a single dynamic analysis run only reveals a small portion of the entire set of behaviors that a malware program could exhibit. With Reanimator, we can automatically generate a model that captures the code that is responsible for the observed behavior. As a result, each execution of a binary contributes a piece to a global knowledge base that stores different instantiations for different behaviors. This knowledge base can then be used to discover dormant functionality in other malware samples.

4.2 System Overview

Reanimator works in three phases, as shown in Figure 4.1. The first two phases are responsible for generating functionality-aware models for different behaviors. The last phase uses previously constructed models to check for dormant behaviors. The following paragraphs outline the three phases in more detail.

4.2.1 Dynamic Behavior Identification

In the first phase, a malware binary is executed in an instrumented, dynamic analysis environment. For this, we obtained access to Anubis [1], a sandbox that is built on top of the whole-system emulator Qemu. Anubis records the invocations of a large set of security-relevant system calls and Windows API functions. In addition, the system uses taint analysis to track data flow dependencies between system and function call arguments.

Based on the output of Anubis, we use a set of specifications to identify different types of interesting, security-relevant behaviors that a malware binary has exhibited during the dynamic analysis. We call such externally-visible, security-relevant behaviors that are observed during dynamic analysis *malware phenotypes*. Examples of phenotypes include sending spam, launching attacks, installing a keyboard logger, and performing password sniffing. To write behavioral specifications for different phenotypes, we build upon previous work that has introduced languages to express specific malware behavior with the help of graphs [10] or automata [14, 24]. Both approaches use as input a trace of system calls observed during dynamic analysis, information that is readily available in the Anubis output.

For our work, we use rules that describe a malware phenotype in terms of the required system or API calls, their arguments, and the data flows between these arguments. This is very similar to *malspecs* [10]. For instance, we detect that a malware program is sending spam by looking for outgoing mail traffic on TCP port 25. Such activity is, by assumption, malicious because Anubis does not support user interaction, and it is very unlikely that a benign program needs to send email without user approval. Similarly, we can detect a network-based attack by matching the data that is provided to the network `send` API call against network intrusion detection signatures. Port scan or denial of service attacks are captured by measuring the frequency of (failed) outgoing connection attempts. Detection of other behaviors requires us to take advantage of data flow information. As an example, a malware "dropper" (or update) behavior is characterized as a data flow from a network socket to a file together with the fact that this file is later executed. In total, we have manually developed nine specifications of high-level behaviors that cover common malware activity. These behaviors are discussed in more detail in Section 4.4. Of course, if needed, the set of patterns can be easily extended to cover additional phenotypes.

The astute reader might wonder why we consider it reasonable to manually write specifications for dynamic behaviors (phenotypes) when we have previously stated that automation is necessary for generating functionality-aware models. The reason is that specifications that operate on dynamic analysis output can capture behavior at a high

level of abstraction. Hence, they are much easier to develop than models that operate on static binary code. This is because with dynamic analysis, one has concrete outputs or events that a specification can be applied to. For example, it is relatively straightforward to identify spam activity by checking for network traffic to port 25 that contains SMTP keywords. On the other hand, it is significantly more difficult to model binary code that is capable of opening a network connection to port 25 and sending out data that conforms to the SMTP specification. Moreover, the same dynamic output can be achieved in many different ways. That is, a *single* phenotype can be implemented by many different code instantiations, each of which might need to be modeled explicitly. Of course, specifications that operate on dynamic output cannot, on their own, achieve Reanimator's main goal, which is precisely to identify those (dormant) behaviors that are *not* executed during dynamic analysis.

Whenever we identify a phenotype B during dynamic analysis, we mark all system calls that are directly related to B. For example, assume that we recognize that a malware sample opens a network connection and sends out a spam mail (by checking that this connection contains SMTP traffic and has destination port 25). In this case, we mark the system call that is responsible for opening the socket (that belongs to the network connection over which the mail was sent), as well as all system calls that write out the mail (spam) data. Similarly, for network sniffing, we would mark the system call that is responsible for opening a promiscuous-mode socket, and all system calls that receive data from this socket. We define the system calls that are marked as related to behavior B the *relevant* system calls for B, and we denote this set as R_B. The set of all relevant system calls $R = \{R_B\}, \forall B$ observed during the dynamic analysis run, serve as the starting point for the next phase.

4.2.2 Extracting Genotype Models

In the second phase, the goal is to locate the part of the binary that is directly responsible for a certain phenotype that was witnessed during the previous dynamic analysis phase. We call the code that is responsible for a particular phenotype a *genotype* for this behavior. Once we have located a genotype, we can build a model for it. The basic idea is that a genotype model can then be leveraged to search for similar code in other binaries.

A main challenge, and a core contribution of this paper, is to develop techniques to find and model genotypes that correspond to behaviors that are seen during a dynamic analysis run. It is important that these genotype models are *precise*, i.e., that they capture only code that is directly responsible for malicious behavior. In particular, a model should not contain parts of shared utility or library routines that are also used

by other functionality. Moreover, genotype models should be *complete*, i.e., they should contain the entire code that is responsible for a particular behavior and not only a fragment. Imprecise or incomplete models can lead to both false negatives or false positives. For example, when a model contains unrelated code, it is possible that this fragment accidentally matches benign code (false positive).

As mentioned previously, the starting point for generating a genotype model is the set of relevant system calls R_B that the previous phase associates with a certain malicious behavior B. We first use a *program slicing step* to identify all instructions that contribute to the input parameters of these system calls, as well as instructions that operate on their output parameters. Typically, the resulting program slices are neither precise nor complete. Thus, we use a subsequent *filtering step* to remove those parts that are not directly related to the observed behavior. Finally, we use a *germination step* to extend the slice to include parts of relevant code that were missed by the initial program slicing step. Typically, these parts are related to instructions that do not directly operate on system call input or output data, but that set up a loop or maintain the program stack. Moreover, the germination step can also include alternative code paths that are part of the dormant functionality but were not executed during the dynamic analysis run. This typically increases the completeness of our genotype model by including code that handles special cases or error conditions that did not occur during the dynamic analysis.

Note that a genotype represents only one instantiation of a particular phenotype. That is, a malware binary might possess a dormant functionality, but our genotype models do not recognize this functionality because the malware binary implements this functionality in a different way (i.e., it has a different genotype for the same phenotype). However, as our empirical results demonstrate, polymorphic variants and code reuse are common and lead to a situation where malware binaries share a significant amount of code. Moreover, whenever a new implementation of a behavior is observed in our sandbox, the system can automatically generate a corresponding genotype model.

4.2.3 Finding dormant functionality

Once we have generated a set of genotype models associated with different malicious behaviors, the third and last step is to use such models to check binaries for dormant functionality. To this end, we statically disassemble an unpacked sample and check for the presence of previously-modeled genotypes. When a code region is found that matches one of our models, we report that this sample contains a dormant functionality that implements the behavior associated with the matching genotype.

Since we use static analysis to identify dormant code in binaries, we need to take into account runtime packing and code obfuscation. To handle packed binaries, we use

a generic unpacking technique similar to previous solutions such as Renovo [17] and OmniUnpack[23]. In general, we envision to use REANIMATOR in combination with a dynamic analysis tool such as Anubis. Thus, it is easy and effective to take a memory snapshot at the end of the dynamic analysis run. Then, we can perform the search for dormant functionality on this unpacked snapshot. The robustness of the system against code obfuscation depends on the concrete implementation choice for the genotype models. As shown in Section 4.4, our current models, which rely on structural information of the binary code, work well and can tolerate differences in the program source code, as well as changes that are the result of different compiler settings or compiler versions. When more robustness is required, one could fall back to semantics-aware code templates or blueprints, although such models incur significantly higher performance costs.

4.3 System Details

In this section, we discuss in more detail our approach to generate genotype models for phenotypes that are identified during dynamic analysis. Then, we discuss how these models can be used to detect dormant functionality.

4.3.1 Genotype Models

As mentioned previously, a genotype is a part of a malware program that is responsible for a particular runtime behavior. Thus, genotype models need to be able to characterize binary code. This can be achieved in different ways. On one end of the spectrum, a model could be implemented as a string of bytes or a sequence of instructions that covers an interesting code section. While such models are fast when searching for dormant functionality, they are very specific. Thus, even minor changes in the malware binary would cause these models to miss relevant code. On the other hand, one could attempt to extract generalized code templates (such as the ones proposed in [9, 11]). While quite robust to semantics-preserving code changes, the detection process using these models is very costly.

For this work, we leverage the techniques proposed in [19] and model code as its corresponding *colored control flow graph (CFG)*. A CFG is a directed graph where nodes are basic blocks, and an edge from node u to v represents a possible control flow (such as a jump or branch) from u to v. The nodes of the CFG we use are colored based on the classes of instructions that are present in the corresponding basic blocks. Instruction classes, as defined in [19], are, for example, “arithmetic,” “logic,” or “data transfer” operations.

CFGs have been used in the past to find similarities between polymorphic worms and malware samples. Also, they have a number of properties that make them particularly useful in our setting. First, focusing on the structure of code instead of instruction sequences makes models robust to simple code insertion and deletion, and to certain classes of code modifications such as register renaming or instruction substitution. Second, using proper optimizations [19], it is fast to search malware programs for code that matches previously-constructed models.

In general, two genotypes are considered similar when their respective CFGs share at least one isomorphic subgraph that is sufficiently large (it has at least k nodes – as in [19], we use $k = 10$). Thus, given a genotype, modeled as a colored CFG G, the problem of finding this genotype in a malware binary is reduced to finding an isomorphic subgraph of size k that is present both in G and in the binary under analysis. Since this is an NP-complete decision problem, previous work [19] introduced an efficient, approximate algorithm. This algorithm generates a subset of all possible k-node subgraphs of G and normalizes them. Each normalized k-node subgraph then serves as a succinct fingerprint of the code region that is modeled. For performance reasons, a hash of the subgraph's normalized representation is typically used. In other words, a genotype model is not the colored CFG itself, but a set of fingerprints that represent it. To search a binary for the presence of a particular genotype, only the fingerprints are used. When one or more fingerprints match, then we assume that the binary contains the corresponding genotype.

4.3.2 Genotype Model Extraction

The goal of the genotype model extraction phase is to map an observed, dynamic behavior (a phenotype) to the code that implements this behavior (the genotype). Once this code is located, we can extract its CFG and generate the corresponding fingerprints. These fingerprints then serve as the genotype model for detecting dormant behaviors in other binaries. The genotype model extraction process operates in three steps, which are discussed in the following.

Program Slicing

The starting point for locating code that is responsible for a particular behavior B is the set of relevant system calls R_B that the dynamic behavior identification phase has found to be associated with B (as discussed in Section 4.2.1).

In a first step, we attempt to find all code that is "related" to the systems calls $r \in R_B$. More precisely, we attempt to find all instructions that either *(i)* compute values that are

used as input parameters to these system calls, or that *(ii)* process the output (return) values from these system calls. The intuition is that when a relevant system call r is part of an observed behavior, then the code responsible for this behavior must either prepare and invoke this system call to produce desired output or use it to obtain necessary inputs that are later processed.

Interestingly, the concept of a set of instructions that are related to a program point is similar to a *program slice* [5]. In general, a program slice consists of all program instructions that affect a given point of interest in the program. In our case, we are interested in all instructions that affect a point of interest through data flow dependencies. That is, our slices only capture data flow between instructions, but we do not include instructions that have an indirect effect through control flow. The point of interest is a system call $r \in R_B$. Thus, a backward slice consists of all instructions such that, for each instruction, there is a data flow from one (or more) of its operands to one of the input arguments of an interesting system call (case *i*). A forward slice, on the other hand, is defined as all instructions for which there is a data flow from the output of an interesting system call to the instruction (case *ii*).

Forward slicing. For certain types of malicious behavior, the malware program needs to process the output of system calls. For example, a malware that implements packet sniffing functionality has to process data that is received via a promiscuous-mode socket, either to log it or to analyze it for specific patterns that are of interest to the attacker (e.g., passwords or credit card numbers). As another example, a program that implements a backdoor has to process the output of the network-related system calls that are used to receive commands from the attacker. To capture the code (genotype) related to such classes of behavior B, we extract a forward slice ϕ, starting from the output parameters of the relevant system calls R_B.

To compute a forward slice starting from the output of a given system call, we leverage the taint information that is provided by Anubis. In addition, we make use of the instruction log that records each operation that the malware under analysis has performed. More specifically, we taint the output of the system call and then include into the slice ϕ all instructions that operate on tainted data (i.e., at least one source operand of an instruction is tainted). Furthermore, we also propagate taint across system calls. That is, we taint the output of system calls when at least one argument of the system call was tainted.

When computing a forward slice, the analysis follows a dynamic approach and directly operates on the instructions that were executed by the malware binary. Because malicious code may be self-modifying, individual instructions cannot simply be identified by their address in the program. Instead, we identify an instruction as a tuple

$\langle address, version_number \rangle$. While the program under analysis is executing, we increment the version number of an instruction whenever the memory at the corresponding address was modified since the last time it was executed.

Backward slicing. While some malware functionality operates on the output of system calls, other behaviors are based on computation that provides inputs to relevant system calls $r \in R_B$. For instance, in the case of a UDP flooding attack, we are interested in how the packet payload and network-related parameters (e.g., ports or destination IP addresses) are determined. Or, in case of spam activity, the interesting part of the program is the genotype that is responsible for setting up a network connection and sending out mails.

To identify the code that is responsible for computing the inputs to relevant system calls, we use a standard dynamic slicing approach [5]. That is, we leverage the instruction and memory access logs that Anubis produces to follow define-use chains backwards, starting from the input parameters of the system calls. More precisely, we start to look for instructions that *define* the values that serve as input to relevant systems calls, and we add these instructions to the slice ϕ. For each of these instructions, we examine their operands and determine the values that they *use*. For each such value, we locate the instruction that defines it, and include it into the slice as well. This process is then continued recursively, adding to ϕ all instructions that define (produce) inputs for instructions already in the slice.

For each system call $r \in R_B$, we compute forward and backwards slices $\phi_{r,forward}$ and $\phi_{r,backwards}$. The output of the program slicing step is the slice ϕ that is the union of all forward and backwards slices:

$$\phi = \bigcup_{r \in R_B} (\phi_{r,forward} \cup \phi_{r,backwards}).$$

Running example. As a running example to illustrate the way in which genotype model generation works, consider the code snippet shown in Program 1. This code snippet shows a part of the main control loop of a bot. In particular, we focus on one *case* statement that is executed when the bot receives a "sniff" command. If the botmaster sends such a command, the program first opens a socket in promiscuous mode (Lines 5-9). Then, for each packet, the code checks whether a TCP packet was received (Lines 11 and 12). If so, the bot scans the packet payload for passwords (indicated by the presence of the string `password`) and logs those that are found (Lines 13-16).

When the code in the example is executed during dynamic analysis, our system would recognize that the `WSAIoctl` call is being used to put a socket in promiscuous mode. This behavior is associated with "sniffing activity," and the dynamic behavior identification

Program 1 Example: Network sniffing

```
1  switch(command){
2  case X:
3    ...
4  case sniff:
5    sock = socket(...);
6    if(sock == INVALID_SOCKET)
7      error();
8    bind(sock,...);
9    WSAIoctl(sock, OPT_PROMISCUOUS, ...);
10   while(recv(sock, buffer,..)){
11     ip = (IPHEADER *)buffer;
12     if (ip->proto == TCP){
13       packet = buffer+sizeof(TCPHEADER);
14       if (strstr(packet, "password") != NULL){
15         write(log, "Got a password!");
16         write(log, packet);
17       }
18     }
19   }
20 }
```

step would mark the `WSAIoctl` call in Line 9 as relevant, together with all other calls operating on the promiscuous socket `sock`; the `socket`, `bind`, and `recv` calls in Lines 5, 8 and 10. The corresponding genotype that we aim to identify and model is the entire *case* statement (the code enclosed in the box), but not the helper functions such as `strstr`.

For packet sniffing activity, the genotype extraction phase first computes a forward and backward slice for all relevant system calls. This includes into the slice Lines 5, 6, 8, 9 and 10 because they operate on the variable `sock`, which is the result of the `socket` system call. Moreover, the system includes the code in Lines 12, 14, and 16, because they operate on the tainted data `buffer` returned by the call to `recv`. Note that Lines 11 and 13 are not (yet) included, because they only manipulate variables that *point* to tainted data, but are not themselves tainted. Furthermore, Line 1 is *not* included in the backwards slice, because our slicing approach takes into account only data flow, and not the effect of control flow decisions.

Filtering

As discussed in the previous section, the program slice ϕ contains all instructions that are connected to relevant system calls by a data flow. However, it is likely that ϕ contains code that is not directly related to the malicious behavior that was observed. This occurs for two main reasons.

First, when generating a forward slice from instructions that operate on system call outputs (tainted data), it is often the case that instructions are included that are part of general purpose utility functions (e.g., string processing routines). This is particularly critical for library functions that are statically compiled into a binary, because models for such functions will match against any code that makes use of the same library.

A second reason why slices often contain code that is not directly related to the observed malware behavior is the fact that backward slices might lead back "too far" in the program. That is, it is not immediately clear when the analysis should stop to follow define-use chains. As a result, slices frequently contain initialization code. Even more problematic, when the malware program is unpacked during runtime, the slice might even include the generic code of the packer. Including such code into the genotype is undesirable because the corresponding model potentially matches all binaries that use the same unpacking routine (which is clearly not related to a certain runtime behavior).

To address the problem of unrelated code that is part of the program slices computed during the first step, we introduce an additional filtering step. The goal of this filtering step is to identify instructions that are not directly responsible for a malicious behavior. To this end, we have developed the following two techniques:

White-listing. The first technique uses white-listing to remove instructions from the

slice ϕ that are not related to behavior B. White-listing requires a set of white-listed genotype models $\Omega = \{\omega\}$. Each white-listed genotype model ω characterizes code that is *not* directly related to a certain behavior B. For each ω, we perform model matching against the program under analysis. The result of model matching is a set of instructions $N_\omega \subseteq N$ of the malware program that successfully matched against ω. We then remove all instructions from ϕ that appear in N_ω.

To obtain a white-list for a malware program, we can make use of the program itself. More precisely, given a number of threads, processes, or distinct executions of a program under analysis, we can include into a white-list for B the genotype model of all the code that is executed by threads or processes that did *not* perform behavior B. That is, whenever a thread or process is run and does not exhibit behavior B, we can include into the white-list for B all code that was executed during this run.

It would also be possible to use "foreign" genotypes for white-listing purposes (where a foreign genotype is derived from programs other than the malware under analysis). For instance, we could assemble a collection of genotype models of standard library functions, or packing routines, and use it to ensure that no such code is included in genotype models of a malware sample. We did not use a foreign white-list in our experiments. However, the results in Section 4.4 show that REANIMATOR nonetheless achieved a high level of accuracy.

Finding exclusive instructions. The second technique relies on identifying instructions that do *not always* operate on tainted data. That is, we identify the set of instructions $\theta \subseteq \phi$ such that all instructions in θ operate on tainted data (output from marked system calls) every time they are executed. We call these instructions *exclusive* to the malware behavior.

The rationale behind exclusive instructions is that code that is directly responsible for a particular behavior is expected to always operate on data that is related to this behavior. General purpose functions, on the other hand, might also be invoked in other contexts. In those contexts, these functions will operate on untainted data, and hence, they will not be included in θ. For example, in the case of packet sniffing, the general-purpose string routines are very likely to be used also by code that is unrelated to manipulating the sniffed packet payloads.

At this point, we could remove all instructions from a slice ϕ that are not an element of θ. However, in certain cases, this limits the possibility of the subsequent germination step to discover additional, relevant instructions. Thus, we perform the subsequent germination step on instructions in ϕ, and use θ only at the end for final post-processing.

Running example. In the example shown in Program 1, the initial slice would not only contain the instructions that are part of the *case* statement, but also the code of

utility functions that are called with tainted data (such as `strstr` in Line 14). Assuming that we have properly white-listed this library routine, the corresponding instructions would be directly removed from the slice. If this code was not white-listed, it would be removed later. The reason is that it likely does not contain exclusive instructions. In contrast, all instructions in the slice that are part of the *case* statement are exclusive (i.e., they are in θ), since they always operate on tainted data.

Germination

A filtered slice ϕ contains instructions that are directly related to an observed, malicious behavior. However, a slice might be incomplete. In particular, a slice might fail to include instructions that are part of a behavior, simply because these instructions do not directly operate on tainted data or because they are not part of define-use chains. Such instructions typically perform auxiliary tasks, for example, saving register values to the stack before a function call, updating a loop counter variable, or performing pointer arithmetic. Others affect the data flow only indirectly, by influencing control flow decisions.

The goal of the germination step is to improve the completeness of a genotype by expanding a slice ϕ to include auxiliary instructions that are also part of the code directly responsible for a behavior. At the same time, we do not want to reduce the precision of a model by including unrelated code.

The basic approach to do this is the following: We consider an instruction as part of the code that implements a behavior when this instruction cannot be executed without executing at least one instruction that is part of ϕ. The intuition behind this approach is that all instructions in a slice are known to be directly related to a certain behavior. Thus, operations that will *only* be executed together with these directly-related instructions should also be considered to be part of this behavior.

Algorithm. More formally, we consider a filtered slice ϕ to be the initial genotype for the corresponding phenotype. In a first step, we add all instructions d to the genotype that are dominated by the slice ϕ. This is a variation of the well-known concept of dominance in graphs. In the traditional case, a node d is dominated by another node n when every path from the start node to d must go through n. In our case, we consider an instruction to be dominated by the slice ϕ when every path from the start node (function entry point in the CFG) to d goes through at least one instruction $n \in \phi$ (but not necessarily the same n for all paths). In a second step, we add all instructions p to the genotype that are post-dominated by the slice ϕ. Again, this is an extension of the traditional concept of post-dominance. In our case, we say that instruction p is post-dominated by slice ϕ when all paths to the exit nodes of the graph, starting at p,

go through at least one $n \in \phi$. As desired, both dominated instructions d and post-dominated instructions p cannot be executed unless at least one instruction $n \in \phi$ is also executed.

To compute dominator and post-dominator relationships, the CFG of the program is needed, and it can be built in one of two ways. First, we can build a *dynamic* CFG from the execution trace, which holds all instructions that were executed during dynamic analysis. This CFG is accurate, since it contains only instructions that were actually executed by the binary under analysis. However, it might be incomplete, since it does not cover program paths that were not executed. To include such paths as well, one can build the *static* CFG by performing an additional, static analysis step that attempts to disassemble the non-executed regions.

In addition to the CFG itself, one requires its start and exit nodes. Currently, we run our analysis on the intra-procedural CFG. Thus, the start node of the graph is the entry point to the function (a new function entry point is recorded whenever our dynamic analysis observes a `call` instruction). The exit nodes of the graph correspond to `return` instructions. This works well when operating on the static CFG. When using a dynamic CFG, however, this approach often misses exit nodes. The reason is that most exit nodes are never executed during the dynamic analysis run. Thus, when using a dynamic CFG, we add *pseudo* exit nodes to all targets of conditional branches that were not executed during dynamic analysis. We currently operate on intra-procedural CFGs for performance and convenience reasons. When malware authors decide to attack our technique, e.g., by splitting their program into a large number of extremely small functions, or by merging all functions into one, our approach can be extended to work on the entire program CFG.

Based on either the dynamic or the static CFG $G = (N, E)$ (N is the set of instructions, and E the control flow edges), the slice ϕ, a start node ϵ, and a set of exit nodes χ, we then apply the algorithm `germinate` (Algorithm 2). The goal of this algorithm is to find additional instructions that should be added to the genotype. To this end, the algorithm first locates and marks all instructions n_ϕ (instructions that are part of slice ϕ) in the program's CFG. Then, it uses graph reachability analysis to identify the instructions that are dominated and post-dominated by the slice ϕ. These instructions are added to ϕ. Since adding instructions to a slice ϕ might increase its dominance and post-dominance in the graph, the algorithm runs in a loop until a fixpoint is reached (Lines 2, 3, and 10).

To find instructions that are dominated by the slice ϕ', the algorithm first removes the nodes that correspond to instructions in ϕ' from the graph. More formally, the algorithm generates an induced subgraph H from the CFG G by removing all nodes from G that are in ϕ' (Line 4). Note that a subgraph H is said to be induced if, for any pair of vertices x and y of H, $x \rightarrow y$ is an edge of H if and only if $x \rightarrow y$ is an edge of G. When the set

Algorithm 2 `germinate()`

Input: The CFG $G = (N, E)$. The slice $\phi = \{n_\phi\}$. The function entry point ϵ and exit points χ.
Result: The extended slice $\psi : \{n_\phi\} \subseteq \{n_\psi\} \subseteq N$.
1: $\phi' \leftarrow \phi$
2: **repeat**
3: $n \leftarrow |\phi'|$
4: $H \leftarrow G[(N \setminus \{n_{\phi'}\})]$
5: $M \leftarrow mark_reachable_forward(H, \epsilon)$
6: $\phi' \leftarrow \phi' \cup (N \setminus M)$
7: $H \leftarrow G[(N \setminus \{n_{\phi'}\})]$
8: $M \leftarrow mark_reachable_backwards(H, \chi)$
9: $\phi' \leftarrow \phi' \cup (N \setminus M)$
10: **until** $|\phi'| = n$
11: $\psi \leftarrow \phi'$

of nodes S in H is a subset of the nodes in G, then we can write $H = G[S]$. Then, on the new graph H, starting from start node ϵ, the algorithm marks all nodes that are still reachable from the start node, following the forward edges (Line 5). All instructions that could not be reached (i.e., all instructions $N \setminus M$ in the graph that are *not* marked) must have been “cut off” from the start node by the previously removed nodes. That is, there is no path from the start node to an unmarked node that does not “pass through” the slice ϕ. As a result, all instructions that correspond to these unmarked nodes are added to the slice (Line 6). A similar approach is used for the post-dominance computation. The only difference is that the mark algorithm starts at the exit nodes χ (Line 7) and follows control flow edges in the opposite direction (backwards, in Line 8).

Model generation. Given the extended slice ψ, the next step is to translate it into a corresponding genotype model. To this end, the system proceeds in two steps. First, it splits the subgraph $G[\psi]$, induced by ψ on the program CFG G, into maximal connected subgraphs $G_1, .., G_J$. It then splits the slice into the corresponding subsets $\psi_1, .., \psi_J$, where ψ_j is the set of instructions corresponding to nodes of G_j. Clearly, $\psi = \bigcup \psi_j$

In the second step, to filter possibly spurious instructions that might have been added by the germination step, we make use of the set of exclusive instructions θ (as introduced in the previous Section 4.3.2). More precisely, we discard all the slices ψ_j that contain no instruction in θ. The final slice is then $\psi_{final} = \{\psi_j | \psi_j \cap \theta \neq \emptyset\}$.

The genotype model γ is defined as the induced subgraph $\gamma = G[\psi_{final}]$.

Running example. The germination phase adds a number of instructions (lines) to the genotype that were not previously considered by the program slicing step. In particular, it adds Lines 11, 13 and 15 (in Program 1), which are dominated by instructions in the slice (for example, by Line 5). This step does not, however, add any instructions outside of the *case* statement. The reason is that there are paths from the start of the function (and the *switch* statement) to other *case* statements that do not traverse any instructions in the slice associated with the sniff behavior.

Interestingly, when we use a dynamic CFG to perform the germination step, then Line 7 would not be considered in the genotype. The reason is that the error condition handled by Line 7 never occurred, so this instruction was never executed, and hence, would not appear in the dynamic CFG at all. If, however, the system uses a static CFG, then this line would be included.

4.3.3 Genotype Matching

The output of the previous step is a set of genotype models γ_B, one for each observed phenotype B. Thus, the system knows, for each genotype model, what the corresponding behavior is. This information can be leveraged to search for dormant functionality.

Initially, the genotype models must be prepared for efficient searching. To this end, as mentioned in Section 4.3.1, each genotype model, which is a graph, is translated into a set of corresponding fingerprints.

To perform genotype matching and, hence, to identify dormant functionality, an unknown binary is first disassembled, and its CFG is extracted. Then, this CFG is searched for the presence of fingerprints (as discussed in the context of polymorphic worms in a previous paper [19]). Whenever a fingerprint matches, we have found the genotype that this fingerprint belongs do. Thus, we know that the malware contains dormant functionality that is capable of producing the runtime behavior that is associated with this genotype.

Since we perform disassembly and control flow extraction of unknown malware samples, we need to overcome the problem of packed executables (according to [6], more than 40% of the samples are packed with a known packer; a number which is likely only a lower bound). To unpack samples, we use a very simple but effective technique. In existing generic unpackers [32, 17], a malware under analysis is first executed in a dynamic malware analysis environment. Since we already execute the analyzed code in Anubis for several minutes, unpacking happens naturally. At the end of the analysis run, we simply take a snapshot of the memory content and perform analysis directly on this dump. This allows us to not only report the results from the dynamic analysis run, but also to report all dormant functionality that was identified.

Our experience showed that this simple unpacking approach worked very well for the malware samples in our evaluation dataset, and it is also sufficient for most contemporary malware that we have encountered. However, we are aware that there are advanced packers that require the use of alternative unpacking techniques [35, 36].

4.4 Evaluation

The goal of the evaluation is to show that REANIMATOR can extract accurate and robust genotype models for a variety of phenotypes. Moreover, we want to demonstrate that these models are capable of efficiently identifying dormant functionality in real-world malware.

4.4.1 Genotype Model Extraction

Phenotypes. To be able to extract genotype models, it is fist necessary to define appropriate phenotypes. To this end, we first specified rules to detect nine phenotypes that correspond to common malware behaviors. Although the following list is clearly not exhaustive, we believe that it is sufficient to demonstrate the flexibility of our approach.

- *spam*: send unsolicited email. This behavior is detected as SMTP traffic at the network level.

- *scan*: perform a port scan. To detect this phenotype, we rely on Anubis' existing network-level portscan detection heuristics.

- *sniff*: perform packet sniffing. This phenotype is detected when a program opens a socket in promiscuous mode.

- *keylog*: log the keys that the user presses. This phenotype is detected when a program invokes one of several Windows API calls that can be used to register callbacks that receive keyboard information.

- *rpcbind*: exploit a Windows DCE/RPC vulnerability over the SMB/CIFS protocol. This is detected at the network level, using appropriate intrusion detection (Snort) signatures.

- *killproc*: kill a process (typically, an anti-virus process). This phenotype is detected when an analyzed program uses a Windows API call to terminate a process that it did not spawn itself.

- *backdoor*: open a back-door. This phenotype is detected when the analyzed program opens and listens on a TCP port.
- *packetflood*: simple denial-of-service. This phenotype is detected when the malware sends more than a certain number of packets per second to a single destination.
- *drop*: "drop" and execute a binary. This behavior is detected when the taint analysis observes a data flow from the network to a file, and this file is later executed.

Genotypes. Using the previously-defined phenotypes, we executed the following four malware samples in our dynamic analysis environment:

- *rbot*: This malware sample is a representative of a classic IRC-based bot. The corresponding source code was available to us. Therefore, we were able to force the bot to connect to our own IRC server, and we instructed it to execute a variety of actions.
- *pushdo*: Pushdo is a sophisticated, modern downloader/dropper Trojan. It connects to a hard-coded list of IP addresses over HTTP and attempts to download and install additional components. We did not have access to Pushdo's source code. Hence, we started the program and allowed it to connect to its command and control infrastructure.
- *cutwail*: Cutwail is a template-based spam engine that is one of the typical payloads of the Pushdo dropper. Initially, we did not have a sample of Cutwail, but we could use Pushdo to download it and run it for us. For details on the Pushdo/Cutwail botnet, we refer the interested reader to [12].
- *allaple*: Allaple [3] is a well-known polymorphic network worm. When started, our variant performs a network scan on TCP ports 135, 139 and 445. Then, it attempts to compromise the services identified by the scan.

We selected these four malware samples because they exhibit (or, in case of *rbot*, they could be instructed to exhibit) a wide range of behaviors in the Anubis sandbox. Also, these samples represent a good mix of a classic and two more advanced bots and a well-known worm.

We then applied Reanimator to the executions of the four malware samples, and automatically extracted ten genotype models. These models are shown in Table 4.1. The table also shows the size of the genotype that was captured by the corresponding model, both in terms of lines of code (when source code was available) and in terms of basic blocks. Note that the number of basic blocks is shown both for the dynamic and

Table 4.1: Lines of Code (where available) and number of basic blocks of genotype extracted in (S)tatic and (D)ynamic mode.

Genotype	**Sample**	**Phenotype**	**LoC**	**Basic Blocks**	
				S	**D**
sniff	rbot	sniff	95	59	31
udpflood	rbot	packetflood	60	51	41
keylog	rbot	keylog	84	59	49
killproc	rbot	killproc	65	42	27
httpd	rbot	backdoor	392	302	236
simplespam	rbot	spam	37	27	26
drop	pushdo	drop	n/a	150	126
spam	cutwail	spam	n/a	532	290
scan	allaple	scan	n/a	99	62
rpcbind	allaple	rpcbind	n/a	333	133

the static CFG extraction approach used during the germination step. As expected, the static approach covers more code (i.e., regions that were not executed during dynamic analysis). Also, it is interesting to observe that a single phenotype (in this case, *spam*) can be implemented in different ways, which results in different genotype models.

4.4.2 Genotype Model Accuracy

In the next step, we wanted to analyze how accurate our extracted genotype models are. To this end, we first examined whether Reanimator is successful in using a particular genotype model to detect the corresponding genotype in other malware binaries. For this analysis, we could make use of a dataset of 208 bot programs that were available to us as source code. This dataset was provided by the authors of [29]. Many of the 208 bots are variants of *rbot*, and indeed, we randomly chose one *rbot* program from this dataset as one of the four malware sample used for genotype model extraction.

Using the source code of this *rbot* sample, we manually extracted code snippets that we considered to be responsible for each of the six observed *rbot* phenotypes (shown in Table 4.1), one code snippet for each behavior. Then, we checked the source of the remaining 207 bot programs for code that is similar to these six code snippets. Of course, even with source code available, manually checking for the presence of similar code in

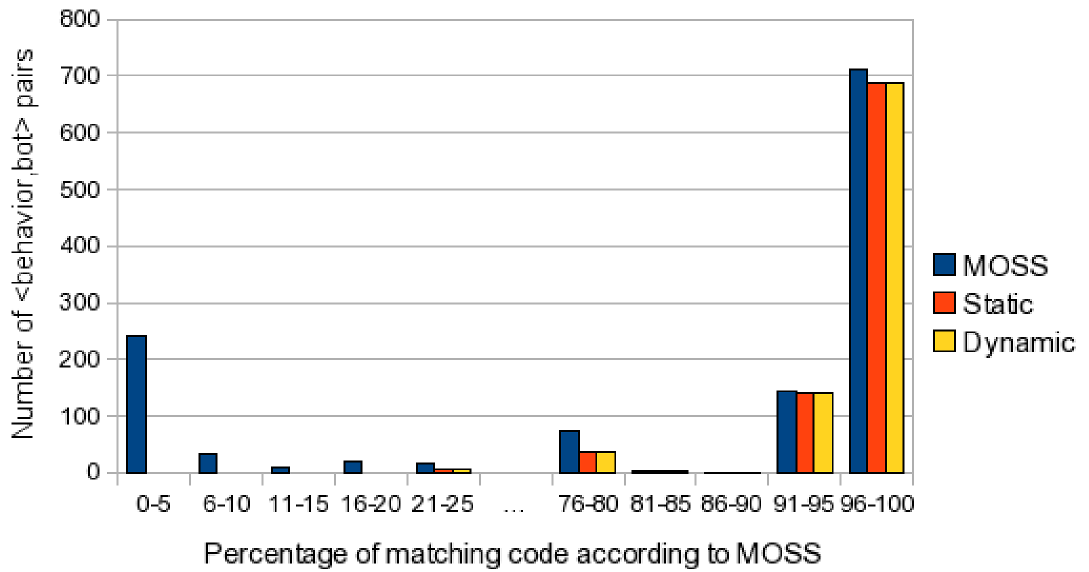

Figure 4.2: Distribution of MOSS scores by percentage, along with the number of matches generated by our genotype models.

hundreds of programs is a tedious task. Therefore, we took advantage of MOSS [34], a free, web-based service for plagiarism detection. MOSS is widely used for detecting plagiarism in computer science classes, and it allowed us to identify those samples that contain code that is similar to one of the manually extracted code snippets.

At this point, we used REANIMATOR to match the six genotype models generated for the single *rbot* instance against the 207 remaining bot binaries. These binaries were obtained by compiling each bot source code using Microsoft Visual Studio 2005. We then compared the matches identified by REANIMATOR with the source code similarity measurements obtained by MOSS. Of course, the hope is that the matches that are independently produced by both techniques have a high overlap. That is, we expect that REANIMATOR reports a certain (dormant) functionality in a malware binary whenever MOSS reports that the corresponding program source contains the code snippet that implements this functionality (or, at least, code that is similar to this snippet).

Figure 4.2 shows a comparison between the matches identified by REANIMATOR and the similarity scores produced by MOSS. For each of the six behaviors j, and for each of the 207 binaries i, MOSS produces a similarity score that indicates the confidence that the code snippet that implements behavior j is present in binary i. We call this similarity score $M_{i,j}$. Figure 4.2 shows a histogram that displays the distribution of the scores $M_{i,j}$. It can be seen that many scores are very high, which confirms the previous observation that the dataset contains many variants of *rbot*.

In addition to the similarity scores for MOSS, Figure 4.2 also contains the results for REANIMATOR. In particular, whenever our technique finds a match for genotype model j in binary i, we first check the similarity score that MOSS reported for this combination, which is $M_{i,j}$. Then, we add 1 to the REANIMATOR results for the bin that corresponds to this score. The intuition is that we expect that whenever our technique reports a match, the corresponding similarity score is high. In other words, we would expect that our system reports a match whenever MOSS' similarity score is high (on the right side of the graph), and nothing when the similarity score is low (on the left side of the graph). The static and dynamic bars for REANIMATOR represent the results achieved by using either a static or a dynamic CFG during the germination step (as discussed in Section 4.3.2). For this dataset, the results are identical. However, as we show, this is not the case on other datasets.

As one can see in Figure 4.2, REANIMATOR's genotype matching results are closely correlated with source code similarity obtained from MOSS. On the right hand side, where the MOSS similarity scores are high, genotype matching is almost invariably successful. On the left hand side, where MOSS produces a low score, there are almost no REANIMATOR matches.

We then manually inspected those cases for which MOSS and REANIMATOR reported

different results. First, we looked at instances where our technique detected a match, but the similarity scores reported by MOSS were low (indicating different code). In particular, we checked the code where MOSS reported low scores (lower than 50%). We found that in all five cases, Reanimator was correct. That is, the sample did indeed implement the functionality that Reanimator found. The low MOSS scores were caused by the fact that large parts of the corresponding source code had been modified, or re-implemented. However, enough of the genotype had been preserved that Reanimator could recognize it. We also inspected the 27 opposite cases where MOSS reported a high similarity, but Reanimator did not find a genotype model match. We found that, in 13 cases, the implementation of the i-th phenotype was present in the j-th bot's source code, but not in its binary. The code was excluded from the build process either at the compilation stage (because of `#ifdef` directives), or at the linking stage, because the linker decided not to include an object that was not required. The remaining 14 cases were false negatives, due to the fact that code was changed to an extent that our models failed to detect the similarity.

Finally, we wanted to verify that the genotype models produced by Reanimator do not match arbitrary binary code. That is, we wanted to understand the risk of false positives produced by our models. To this end, we used our ten genotype models and applied them to a dataset that consisted of 1,949 files found in the `system32` directory of a Windows XP installation. Of course, we do not expect any of our genotypes to match on benign Windows program. Indeed, no matches were found.

4.4.3 Robustness

In this section, we evaluate the effects of different compilers and optimizations on Reanimator's accuracy. Specifically, we aim to test whether a genotype model extracted from a binary can be successfully matched against binaries that were compiled with *different* compiler versions or optimization options. For this, we use the same dataset of 208 bot sources, and the same genotype models discussed in the previous section.

As a first test, we re-compiled the bot sources using the same compiler (Microsoft Visual Studio 2005), but with different optimization and inlining options. The results are summarized in Table 4.2. The table shows that for all the genotype models, except for *simplespam*, different compiler options have a very limited effect on the Reanimator results. The number of matching binaries are reduced by less than 7%. The *simplespam* genotype model is more brittle. This is because the *rbot* sample implements this functionality in only 37 lines of code, as opposed to 60 to 392 lines for the other behaviors. Clearly, code re-use is easier to detect when larger code fragments are involved.

For the second test, we re-compiled ten of the 208 bot samples using different versions

Table 4.2: Number of samples (out of 208) matching each behavioral model in (S)tatic and (D)ynamic mode, using different compilation options.

Compiler options	httpd		keylog		killproc		simplespam		udpflood		sniff	
	S	**D**	**S**	**D**	**S**	**D**	**S**	**D**	**S**	**D**	**S**	**D**
No parameters	144	144	133	133	149	149	135	135	136	136	129	129
Minimize Size	154	154	133	133	158	158	158	158	138	138	134	134
Optimize for Speed	144	144	133	133	149	149	1	1	136	128	129	124
Full Optimization	144	144	133	133	149	149	1	1	136	128	129	124
Only __inline	144	144	133	133	149	149	135	135	136	136	129	129
Any suitable	144	144	133	133	149	149	135	135	136	136	129	129

Table 4.3: Number of samples (out of 10) matching each behavioral model in (S)tatic and (D)ynamic mode, using different compilers.

Compiler	httpd		keylog		killproc		simplespam		udpflood		sniff	
	S	**D**	**S**	**D**	**S**	**D**	**S**	**D**	**S**	**D**	**S**	**D**
VS 2003	10	10	10	10	10	10	10	10	10	10	10	10
VS 2005	10	10	10	10	10	10	10	10	10	10	10	10
VS 2008	10	10	10	10	10	10	10	10	10	10	10	10
Intel	10	0	0	0	0	0	0	0	0	0	0	0

of the Visual Studio compiler, as well as the Intel C++ Compiler Professional Edition 11.1. We restricted this test to ten samples because we were not able to completely automate the compilation process with different compilers. More precisely, we learned that different compilers accept slightly different dialects of C source code, and hence, source code needed to be adapted to be accepted by a different compiler.

As can be seen in Table 4.3, Reanimator is robust to different compiler versions, but mostly fails to match genotypes in binaries produced by a completely different compiler. Nonetheless, our results compare favorably to the state-of-the-art work on binary clone detection [33]. Results from [33] show false negative rates of over 96% on identical functions when simply changing compiler options.

While a malware author could still attempt to evade our tool by re-compiling malware with different compilers, this only allows him to generate a limited number of variants. To correctly match against all samples, Reanimator would simply need to generate a genotype model for each variant.

4.4.4 Genotype Matching Results

In this section, we discuss Reanimator's effectiveness on four real-world datasets:

- *irc_bots*: This dataset consists of 10,238 binaries that performed IRC traffic when analyzed in Anubis, and are, therefore, likely to be IRC-based bot samples. Furthermore, these samples were selected based on the output of the SigBuster tool for *not* being packed with a known packer.

- *packed_bots*: This dataset is similar to the *irc_bots* dataset. It consist of 4,523 binaries that perform IRC traffic during Anubis analysis. SigBuster was able to recognize that these samples are packed.

- *pushdo*: This dataset consists of 77 pushdo binaries. To identify Pushdo, we relied on anti-virus signatures to select 25 samples. We also used a known, characteristic behavior of the Pushdo sample observed during analysis in Anubis to identify another 52 samples. Specifically, Pushdo samples request updates by sending an HTTP query for a URL that starts with the string `/40E8`.

- *allaple*: This dataset consists of 64 Allaple samples, identified using anti-virus signatures.

Table 4.4 shows the results of matching the six genotype models extracted from the rbot sample against the two datasets of IRC bots. To compare the coverage provided

Table 4.4: Genotype matching results on IRC datasets.

Genotype	**Phenotype**	**irc_bots**				**packed_bots**			
		B	**S**	**D**	**B ∩ S**	**B**	**S**	**D**	**B ∩ S**
httpd	backdoor	2014	636	635	279	840	425	425	264
keylog	keylog	0	293	254	0	0	120	111	0
killproc	killproc	0	400	400	0	4	62	62	0
simplespam	spam	154	409	409	0	53	204	204	0
udpflood	packetflood	0	374	342	0	0	139	122	0
sniff	sniff	43	270	72	0	120	204	45	0

Table 4.5: Genotype matching results on pushdo and allaple datasets.

Genotype	**pushdo**				**allaple**			
	B	**S**	**D**	**B ∩ S**	**B**	**S**	**D**	**B ∩ S**
drop	50	54	54	46	0	0	0	0
spam	1	43	42	1	0	0	0	0
scan	23	0	0	0	58	61	61	58
rpcbind	5	9	0	1	62	61	61	58

by our tool with the results reported by dynamic analysis approaches, we show, for each genotype, the detection results for the corresponding phenotype based on a single execution in Anubis. The observed, dynamic behaviors in Anubis are shown in columns marked with B.

Comparing the Anubis results with the static and dynamic REANIMATOR results shows that, even with a limited set of genotypes derived from a single malware binary, our techniques can dramatically improve the number of malware capabilities that are discovered. For instance, for the sniffing behavior, Anubis detects 163 samples, while REANIMATOR detects 474. For some of the other genotypes, the increase in coverage is even more significant. For example, only four binaries killed another process while executed in Anubis.

The $B \cap S$ column shows the number of samples matched by both REANIMATOR and Anubis. That fact that, for most behaviors, no samples were matched by both Anubis and Reanimator may seem surprising at first. However, it does not indicate false negatives on REANIMATOR's part. The genotype models detected by REANIMATOR correspond to a single implementation of a certain behavior. Hence, completely unrelated implementations would require additional models. The samples that included the *simplespam* phenotype according to REANIMATOR, for instance, did not send spam during Anubis analysis. This indicates that, during analysis, the bot did not receive commands triggering this behavior. The likely reason is that the spam functionality modeled by the *simplespam* genotype is very primitive compared to modern, template-based spam engines. As a result, it is rarely (or never) used.

Table 4.5 shows results on the *pushdo* and *allaple* datasets. For some genotypes, REANIMATOR does not provide a significant additional coverage. These genotypes correspond to behaviors that are performed by the malware every time it runs, such as dropping a payload by Pushdo. Nevertheless, we gain some additional coverage (8 samples) in cases where a functionality was not successfully executed in Anubis. In this specific case, this is because the Pushdo command and control servers could not be reached. For the *spam* behavior, REANIMATOR provides a significant increase in coverage.

4.4.5 Performance

In this section, we briefly discuss the performance of the REANIMATOR genotype model extraction and genotype model matching techniques.

- *Genotype model extraction*: Performing genotype model extraction on a single, five-minute execution of a binary in Anubis required under two minutes on standard desktop

hardware. Thus, it is feasible in practice to integrate model extraction into the workflow of a large-scale malware analysis system such as Anubis.

- *Genotype model matching*: The genotype model matching techniques used by REANIMATOR are efficient. Matching against the entire 2.5GB *irc_bots* dataset (with more than ten thousand samples) took $2,511$ seconds in total on standard desktop hardware. Hence, around .25 seconds were spent on each binary. This performance is negligible compared to the cost of dynamic analysis.

4.4.6 Limitations

In our experiments, we have seen that our genotype matching technique is successful in statically analyzing real-world malware code and finding dormant functionality. However, malware authors could make it more difficult for REANIMATOR to perform this matching step. For this, they could develop evasion techniques specifically targeted at our tool, such as semantics-preserving obfuscation of the control flow graph. As an example, they could intersperse their program's entire CFG with a large number of spurious nodes and edges. Such techniques could be countered by performing genotype model matching using more powerful, semantic-aware models [31].

On the other hand, malware authors are more likely to prefer generic evasion techniques that are capable of defeating a wide range of analysis and detection approaches. This goal can be achieved by advanced packing techniques (such as emulation-based packing and conditional code obfuscation) that cannot be easily defeated using generic unpacking methods. While recent work has addressed emulation-based packing [35], conditional code obfuscation [36] can, in certain settings, provide strong guarantees that the code cannot be analyzed statically. This is a limitation that REANIMATOR, or any other tool relying on static analysis, faces.

Another limitation of our approach is that, to generate a genotype model, REANIMATOR needs to observe the execution of the corresponding behavior in at least one dynamic analysis run. Therefore, behavior that is *never* executed inside our sandbox cannot be detected. As a consequence, similar to other dynamic analysis approaches, REANIMATOR can be defeated by malware that detects the analysis sandbox and refuses to run. Furthermore, it cannot detect time- or logic- bombs until they are triggered by at least one sample. However, it may be possible to combine our technique with other approaches for improving the coverage of dynamic analysis, such as multiple path exploration [26]. Using REANIMATOR, the insight provided by applying such computationally-expensive techniques to a single malware execution could be leveraged to provide information for other samples.

5 Conclusions

In order to respond to the increasing number of new samples of malware created daily, novel techniques to quickly assess whether or not a given sample is new and worthy of analysis are needed.

In the context of WOMBAT we dedicated some effort to low-cost techniques that could allow to cope with the large number of malware instances produced by polymorphic techniques and to easily distinguish them from new malware variants.

In particular, as reported in D11 and in this deliverable, we experimented with structural information on the malware samples to assess the novelty of a given sample and obtain a preliminary analysis of it.

As reported in Chapter 2, malware can be clustered according to static, structural features. We actually developed two approaches, one solely based on static characteristics that can be inferred by the analysis of the Portable Executable (PE) format headers of Windows executables; the other based on the pairwise difference between all samples. Both have given interesting and promising results, which will also impact further deliverables of the project.

Chapter 3 described how Argos was extended for structural analysis of the shellcode collected during an attack. This extension is able to analyze shellcode and extract the different layers of unpacking that it employs and the final "real" shellcode that performs useful actions for the attacker.

Finally, Chapter 4 demonstrates how we can combine dynamic and static analysis to identify dormant functionality in malware. The main insight behind our system is that we can leverage a single dynamic observation of malicious behavior in one malware sample to statically detect the same functionality in other malware programs. Thus, we can significantly increase our knowledge about the capabilities of malicious code when compared to the results delivered by dynamic analysis alone. Our experiments demonstrated that the generated models accurately capture code parts (genotypes) that are responsible for a diverse set of malicious behaviors. Moreover, they showed that the system, which we called REANIMATOR, can significantly increase the coverage of dynamic analysis systems.

In conclusion, our experiences confirmed that structural information can be used to obtain a preliminary clustering of malware in families, to analyze and unpack the shellcode used in an attack, and to statically seek for a specific interesting malware behavior

which has been dinamically observed in a given sample.

Bibliography

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

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

[3] F-Secure Malware Information Pages - Allaple.A. http://www.f-secure.com/v-descs/allaple_a.shtml, 2008.

[4] Virus Total. http://www.virustotal.com/, 2008.

[5] H. Agrawal and J. Horgan. Dynamic Program Slicing. In *Conf. on Programming Language Design and Implementation (PLDI)*, 1990.

[6] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. Insights into current malware behavior. In *LEET'09: 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats, April 21, 2009, Boston, MA, USA*, Apr 2009.

[7] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. 2008.

[8] E. Carrera. Pefile, http://code.google.com/p/pefile/.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, 2003.

[10] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 5–14, New York, NY, USA, 2007. ACM.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symp. on Security and Privacy*, pages 32–46, 2005.

[12] A. Decker, D. Sancho, L. Kharouni, M. Goncharov, and R. McArdle. A study of the Pushdo / Cutwail Botnet. `http://us.trendmicro.com/imperia/md/content/us/pdf/threats/securitylibrary/study_of_pushdo.pdf`, 2009.

[13] F-Secure. Malware information pages: Allaple.a, `http://www.f-secure.com/v-descs/allaplea.shtml`.

[14] G. Jacob, H. Debar, and E. Filiol. Malware behavioral detection by attribute-automata using abstraction from platform and language. In *Symp. on Recent Advances in Intrusion Detection (RAID)*, 2009.

[15] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review, 1999.

[16] A. K. Jain, E. Topchy, M. H. C. Law, and J. M. Buhmann. Landscape of clustering algorithms. In *In Proceedings of the 17th International Conference on Pattern Recognition (ICPR*, pages 260–263, 2004.

[17] M. G. Kang, P. Poosankam, and H. Yin. Renovo: a hidden code extractor for packed executables. In *WORM '07: Proceedings of the 2007 ACM workshop on Recurring malcode*, 2007.

[18] H.-A. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. In *USENIX Security Symposium*, 2004.

[19] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, 2005.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *USENIX Symp. on Operating System Design and Implementation (OSDI)*, 2004.

[21] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM conference on Computer and communications security*, pages 290–299. ACM New York, NY, USA, 2003.

[22] J. Macdonald. Versioned file archiving, compression, and distribution. Technical report, 1999.

[23] L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conf. (ACSAC)*, 2007.

[24] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Symp. on Recent Advances in Intrusion Detection (RAID)*, 2008.

[25] McAfee. Threats Report First Quarter 2009, 2009.

[26] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symp. on Security and Privacy*, 2007.

[27] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[28] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, 2005.

[29] J. Oberheide, M. Bailey, and F. Jahanian. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[30] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an Emulator for Fingerprinting Zero-Day Attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[31] 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.

[32] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, 2006.

[33] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *18th Int. Symp. on Software testing and analysis (ISSTA)*, 2009.

[34] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[35] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *IEEE Symp. on Security and Privacy*, 2009.

[36] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[37] Symantec. Global Internet Security Threat Report: Trends for 2008, April 2009.

[38] J. Wilhelm and T. Chiueh. A Forced Sampled Execution Approach to Kernel Rootkit Identification. In *Symp. on Recent Advances in Intrusion Detection (RAID)*, 2007.

[39] T. Yetiser. Polymorphic viruses - implementation, detection, and protection , `http://vx.netlux.org/lib/ayt01.html`.