Identifying Dormant Functionality in Malware Programs

Paolo Milani Comparetti*, Guido Salvaneschi†, Engin Kirda‡,
Clemens Kolbitsch*, Christopher Kruegel§ and Stefano Zanero†
*Technical University of Vienna
{pmilani,kolbitsch}@seclab.tuwien.ac.at
†Politecnico di Milano
{salvaneschi,zanero}@elet.polimi.it
‡Institut Eurecom
kirda@eurecom.fr
§University of California, Santa Barbara
chris@cs.ucsb.edu

***Abstract*—To handle the growing flood of malware, security vendors and analysts rely on tools that automatically identify and analyze malicious code. Current systems for automated malware analysis typically follow a dynamic approach, executing an unknown program in a controlled environment (sandbox) and recording its runtime behavior. Since dynamic analysis platforms directly run malicious code, they are resilient to popular malware defense techniques such as packing and code obfuscation. Unfortunately, in many cases, only a small subset of all possible malicious behaviors is observed within the short time frame that a malware sample is executed. To mitigate this issue, previous work introduced techniques such as multi-path or forced execution to increase the coverage of dynamic malware analysis. Unfortunately, using these techniques is potentially expensive, as the number of paths that require analysis can grow exponentially.**

In this paper, we propose REANIMATOR, a novel solution to determine the capabilities (malicious functionality) of malware programs. Our solution is based on the insight that we can leverage behavior observed while dynamically executing a specific malware sample to identify similar functionality in other programs. More precisely, when we observe malicious actions during dynamic analysis, we automatically extract and model the parts of the malware binary that are responsible for this behavior. We then leverage these models to check whether similar code is present in other samples. This allows us to statically identify *dormant functionality* (functionality that is not observed during dynamic analysis) in malicious programs. We evaluate our approach on thousands of real-world malware samples, and we show that our system is successful in identifying additional, malicious functionality. As a result, our approach can significantly improve the coverage of malware analysis results.

I. INTRODUCTION

Malware is a significant problem and the root cause for many security threats on the Internet. For each new malware binary that is discovered, it is important to understand its malicious capabilities, its propagation vectors, and its impact on the local system. This is necessary to determine the type and severity of the threat that the malware poses. Also, this information is valuable to create detection signatures and removal procedures. Of course, given the sheer volume of new samples that appear every day, obtaining a preliminary understanding of the capabilities of a malware binary requires the use of automated analysis systems.

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 [3]–[5] 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) [3], [4], or simply force the execution along a different path [5]. 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.

In this paper, we propose 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 [6], 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.

We performed empirical experiments to demonstrate the accuracy of our system by comparing it with the results of a source-code-level plagiarism detection tool. Furthermore, we tested our system on large, real-world malware datasets. Our results show that REANIMATOR can successfully detect dormant functionality and significantly increase the coverage of dynamic analysis techniques.

The main contributions of this paper are the following:

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

II. 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 [7], token sequences [8], or control flow graphs [9] 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 [10] and malware blueprints [11]. 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 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.

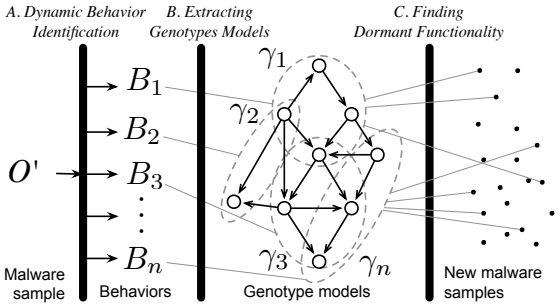

Figure 1. An overview of the REANIMATOR workflow.

III. SYSTEM OVERVIEW

REANIMATOR works in three phases, as shown in Figure 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.

A. 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 [12] or automata [13], [14]. 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* [12]. 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 V. 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.

B. 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.

C. 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 [15] and OmniUnpack [16]. 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 V, 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.

IV. 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.

A. 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 [10], [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 [9] 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 [9], 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 [9], 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 [9], 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 [9] 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.

B. 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.

1) **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 III-A).

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* [17]. 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 [17]. 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:

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

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 }
```

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 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.

2) **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 V 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.

3) **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.

Algorithm 1 `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:  φ' ← φ
2:  repeat
3:      n ← |φ'|
4:      H ← G[(N \ {n_φ'})]
5:      M ← mark_reachable_forward(H, ε)
6:      φ' ← φ' ∪ (N \ M)
7:      H ← G[(N \ {n_φ'})]
8:      M ← mark_reachable_backwards(H, χ)
9:      φ' ← φ' ∪ (N \ M)
10: until |φ'| = n
11: ψ ← φ'
```

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 1). 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 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 $N \backslash M$ 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 IV-B2). 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.

C. 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 IV-A,

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 [9]). 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 [18], 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 [15], [19], 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 [20], [21].

V. 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.

A. 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 [22].
- *allaple*: Allaple [23] 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 I. The table also shows the size of the genotype that was captured by the corresponding model, both in terms of lines of code (when

Table I
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

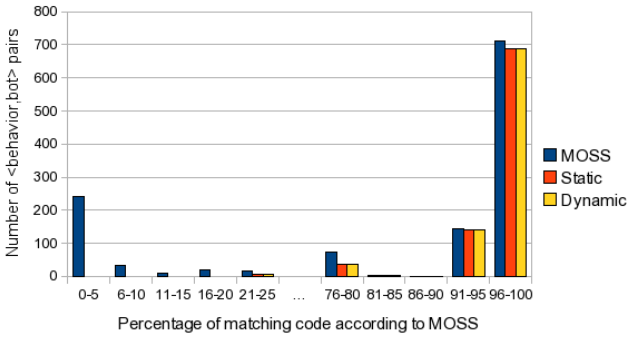

Figure 2. Distribution of MOSS scores by percentage, along with the number of matches generated by our genotype models.

source code was available) and in terms of basic blocks. Note that the number of basic blocks is shown both for the dynamic and 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.

B. 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 [24]. 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 I), 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 hundreds of programs is a tedious task. Therefore, we took advantage of MOSS [25], 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 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 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 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 IV-B3). 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 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.

C. 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 II. 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 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 III, 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 [26]. Results from [26] 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.

D. 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 IV 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 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

Table II
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 III
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

Table IV
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 V
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

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 V 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.

E. 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.

F. 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 [27].

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 [20], conditional code obfuscation [21] 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 [4]. 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.

VI. RELATED WORK

A number of related works have explored static and dynamic approaches to analyze malware samples. As we observed in Section I, one of the motivations to develop REANIMATOR is that dynamic analysis suffers from the weakness of partial coverage. To counter this, previous research has proposed either to run samples multiple times by crafting inputs that invert the outcomes of conditional branches (possible triggers) [3], [4] or simply to force execution along a different path [5]. Unfortunately, using such techniques is potentially expensive, as the number of paths that require analysis can grow exponentially. The path explosion problem similarly affects dynamic software testing systems such as EXE [28], DART [29], and SAGE [30], which use symbolic execution to execute more code paths for the purpose of finding more bugs.

Structural characteristics, and in particular CFGs, have been extensively used in static analysis. Besides the results in [9], which we leveraged in this work, a similar approach is presented in [31]. The difference is that the authors of [31] apply normalization techniques to reduce the effects of well-known code mutation techniques. After normalization, they use inter-procedural CFGs, linking together the CFGs of each function of a program.

Structural code characteristics have also been used to recognize similarities between program binaries. For example, the authors of [32] and [33] both define a distance function over a set of malware samples that is based on their function call graphs. In [32], the authors propose using such a distance function to classify malware and to create a phylogeny of malware binaries. SMIT [33] also uses properties of the function call graphs to implement an efficient nearest-neighbor search on large malware datasets. Function call graphs can provide an overall view of an entire binary, and are, therefore, suitable for globally comparing binaries. However, these techniques are typically not well-suited for detecting the presence of a small code fragment in a larger program. Moreover, they do not associate any semantics (information about phenotypes) with call graphs.

Besides using control flow or instruction graphs, other formalisms have been used to identify viral code sequences. In [11], model checking is used to identify parts of a program that implement a previously specified malicious code template. This combats common obfuscation techniques.

The technique was later extended in [27], allowing more general code templates and using advanced static analysis techniques. Model checking is used also in [34] to semantically identify malware using a temporal logic specification. All of these systems share the limitation that models of each behavior instantiation have to be specified manually. REANIMATOR, in contrast, only requires generic behavioral models (phenotypes) of what constitutes a malicious behavior, and is then capable of automatically identifying code that implements that behavior (the genotype). Furthermore, these techniques are largely orthogonal to our work. Once REANIMATOR has identified a genotype, it could potentially make use of some of these more sophisticated (and computationally costly) detection techniques for genotype matching.

Similar to our system, AGIS [35] uses a combination of dynamic and static techniques to analyze malware. More precisely, the system uses static analysis to identify instruction sequences that lead to system calls that are associated with malicious activity. These instruction sequences are then used as infection (detection) signatures. A significant difference to our system is the fact that AGIS simply includes into signatures all instructions that lead to interesting system calls. Our system, on the other hand, attempts to extract only those parts of the code that are actually responsible for the observed behavior. This is achieved by REANIMATOR's filtering and germination steps.

Plagiarism detection through source code analysis is a well-explored theme. Besides MOSS [25], other significant work in this area includes CCFinder [36], Dup [37], and CP-Miner [6]. Some tools are based on textual analysis, comparing the lines of code. Others compare the token sequences of lines [36], [37], or tree-representations of the code [38]. Others, more interestingly, use the program dependency graphs of code (i.e., control and data flow dependencies of functions) [39], something conceptually close to the use of CFGs on assembly code. Plagiarism detection on binaries is less developed. In [26], the authors have extended a previous system [38] to work on assembly instructions, through some normalization techniques and devising novel models to compactly represent information related to binary instructions. Again, no semantic information is associated with detected clones, and the approach is much less robust than ours with regards to small code changes, for example, due to different compiler options (as discussed in Section V).

VII. CONCLUSIONS

Dynamic malware analysis systems provide important information about the capabilities of malicious code found in the wild. However, they typically only execute a single execution path. As a result, these systems often fail to observe a significant fraction of the entire functionality that a malware sample implements.

In this paper, we present REANIMATOR, a novel system to identify dormant functionality in malware. The main insight behind our system is that we can leverage a single observation of malicious behavior in one malware sample to detect the same functionality in other malware programs (even when they do not exhibit this behavior). In order to achieve this goal, our system operates in three steps. First, we use simple rules to identify security-related behavior (phenotypes) in the output and the events that a malware sample produces during dynamic analysis. Second, we automatically locate and model the code (genotype) that is responsible for this behavior. Third, we reuse previously-generated genotype models for a specific behavior to statically detect similar code in malware that does not exhibit this behavior during the dynamic analysis.

Our approach allows us to unveil dormant functionality in 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 demonstrate that the generated models accurately capture code parts (genotypes) that are responsible for a diverse set of malicious behaviors. Moreover, they show that REANIMATOR can significantly increase the coverage of dynamic analysis systems.

ACKNOWLEDGMENTS

The authors would like to thank Michael Bailey and Jon Oberheide for providing us the bot source code samples used in our evaluation, as well as Thorsten Holz and Morgan Marquis-Boire for help with the unpacking of malware. We also acknowledge the help of Federico Maggi with the artwork, and proofreading. This work has been partially supported by the European Commission through project IST-216026-WOMBAT funded under the 7th framework program, by the ONR under grant no. N000140911042 and by the National Science Foundation (NSF) under grant no. 0845559.

REFERENCES

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

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

[3] 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.

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

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

[6] 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.

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

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

[9] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Symp. on Recent Advances in Intrusion Detection (RAID)*, 2005.

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

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

[12] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *6th Joint European software engineering conf. and ACM SIGSOFT Symp. on Foundations of Software Engineering (ESEC-FSE)*, 2007.

[13] 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.

[14] 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.

[15] M. G. Kang, P. Poosankam, and H. Yin, "Renovo: a hidden code extractor for packed executables," in *ACM Workshop on Recurring malcode (WORM)*, 2007.

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

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

[18] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "Insights into current malware behavior," in *2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[19] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the hidden-code extraction of unpack-executing malware," in *22nd Annual Computer Security Applications Conf. (ACSAC)*, 2006.

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

[21] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Network and Distributed System Security (NDSS)*, 2008.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in *Principles of Programming Languages (POPL)*, 2007.

[28] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 1–38, 2008.

[29] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *ACM SIGPLAN Conf. on Programming language design and implementation*, 2005.

[30] P. Godefroid, M. Levin, and D. Molnar, "Automated Whitebox Fuzz Testing," in *15th Symp. on Network and Distributed System Security (NDSS)*, 2008.

[31] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[32] E. Carrera and G. Erdelyi, "Digital genome mapping–advanced binary malware analysis," in *Virus Bulletin Conf.*, 2004, pp. 187–197.

[33] X. Hu, T. Chiueh, and K. Shin, "Large-scale malware indexing using function-call graphs," in *ACM Conf. on Computer and Communications Security (CCS)*, 2009.

[34] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[35] Z. Li, X. Wang, Z. Liang, and M. K. Reiter, "Agis: Towards automatic generation of infection signatures," in *DSN*, 2008.

[36] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code," *IEEE Trans. on Soft. Eng.*, pp. 654–670, 2002.

[37] B. Baker, "On finding duplication and near-duplication in large software systems," in *Working Conf. on Reverse Engineering (WCRE)*, 1995.

[38] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th Int. Conf. on Software Engineering (ICSE)*, 2007.

[39] J. Krinke, "Identifying similar code with program dependence graphs," in *8th Working Conf. on Reverse Engineering (WCRE)*, 2001.