Polymorphic Worm Detection Using Structural Information of Executables

Christopher Kruegel[1], Engin Kirda[1], Darren Mutz[2], William Robertson[2], and Giovanni Vigna[2]

[1] Technical University of Vienna
chris@auto.tuwien.ac.at,
engin@infosys.tuwien.ac.at
[2] Reliable Software Group,
University of California, Santa Barbara
{dhm, wkr, vigna}@cs.ucsb.edu

Abstract. Network worms are malicious programs that spread automatically across networks by exploiting vulnerabilities that affect a large number of hosts. Because of the speed at which worms spread to large computer populations, countermeasures based on human reaction time are not feasible. Therefore, recent research has focused on devising new techniques to detect and contain network worms without the need of human supervision. In particular, a number of approaches have been proposed to automatically derive signatures to detect network worms by analyzing a number of worm-related network streams. Most of these techniques, however, assume that the worm code does not change during the infection process. Unfortunately, worms can be *polymorphic*. That is, they can mutate as they spread across the network. To detect these types of worms, it is necessary to devise new techniques that are able to identify similarities between different mutations of a worm.

This paper presents a novel technique based on the structural analysis of binary code that allows one to identify structural similarities between different worm mutations. The approach is based on the analysis of a worm's control flow graph and introduces an original graph coloring technique that supports a more precise characterization of the worm's structure. The technique has been used as a basis to implement a worm detection system that is resilient to many of the mechanisms used to evade approaches based on instruction sequences only.

Keywords: Network worms, Polymorphic code, Structural analysis, Intrusion detection.

1 Introduction

In recent years, Internet worms have proliferated because of hardware and software mono-cultures, which make it possible to exploit a single vulnerability to compromise a large number of hosts [25].

Most Internet worms follow a scan/compromise/replicate pattern of behavior, where a worm instance first identifies possible victims, then exploits one or more vulnerabilities to compromise a host, and finally replicates there. These actions are performed through network connections and, therefore, network intrusion detection systems (NIDSs) have been proposed by the security community as mechanisms for detecting and responding to worm activity [16, 18].

However, as worms became more sophisticated and efficient in spreading across networks, it became clear that countermeasures based on human reaction time were not feasible [23]. In response, the research community focused on devising a number of techniques to automatically detect and contain worm outbreaks.

In particular, the need for the timely generation of worm detection signatures motivated the development of systems that analyze the contents of network streams to automatically derive worm signatures. These systems, such as Earlybird [19] and Autograph [6], implement a *content sifting* approach, which is based on two observations. The first observation is that some portion of the binary representation of a worm is invariant; the second one is that the spreading dynamics of a worm is different from the behavior of a benign Internet application. That is, these worm detection systems rely on the fact that it is rare to observe the same byte string recurring within network streams exchanged between many sources and many destinations. The experimental evaluation of these systems showed that these assumptions hold for existing Internet worms.

A limitation of the systems based on content sifting is the fact that strings of a significant length that belong to different network streams are required to match (for example, byte strings with a length of 40 bytes are used in [19]). Unfortunately, the next generation of Internet worms is likely to be *polymorphic*. Polymorphic worms are able to change their binary representation as part of the spreading process. This can be achieved by using self-encryption mechanisms or semantics-preserving code manipulation techniques. As a consequence, copies of a polymorphic worm might no longer share a common invariant substring of sufficient length and the existing systems will not recognize the network streams containing the worm copies as the manifestation of a worm outbreak.

Although polymorphic worms have not yet appeared in the wild, toolkits to support code polymorphism are readily available [5, 11] and polymorphic worms have been developed for research purposes [7]. Hence, the technological barriers to developing these types of Internet worms are extremely low and it is only a matter of time before polymorphic worms appear in the wild.

To detect this threat, novel techniques are needed that are able to identify different variations of the same polymorphic worm [15]. This paper presents a technique that uses the structural properties of a worm's executable to identify different mutations of the same worm. The technique is resilient to code modifications that make existing approaches based on content sifting ineffective.

The contributions of this paper are as follows:

– We describe a novel fingerprinting technique based on control flow information that allows us to detect structural similarities between variations of a polymorphic worm.

- We introduce an improvement of the fingerprinting technique that is based on a novel coloring scheme of the control flow graph.
- We present an evaluation of a prototype system to detect polymorphic worms that implements our novel fingerprinting techniques.

This paper is structured as follows. Section 2 discusses related work. Section 3 presents the design goals and assumptions of our fingerprinting technique and provides a high-level overview of the approach. In Section 4, we describe how the structure of executables is extracted and represented as control flow graphs. In Section 5, we discuss how fingerprints are generated from control flow graphs, and we present an improvement of our scheme that is based on graph coloring. In Section 6, a summary of the actual worm detection approach is given. Section 7 evaluates our techniques, and in Section 8, we point out limitations of the current prototype. Finally, Section 9 briefly concludes.

2 Related Work

Worms are a common phenomenon in today's Internet, and despite significant research effort over the last years, no general and effective countermeasures have been devised so far. One reason is the tremendous spreading speed of worms, which leaves a very short reaction time to the defender [22, 23]. Another reason is the distributed nature of the problem, which mandates that defense mechanisms are deployed almost without gap on an Internet-wide scale [14].

Research on countermeasures against worms has focused on both the detection and the containment of worms. A number of approaches have been proposed that aim to detect worms based on network traffic anomalies. One key observation was that scanning worms, which attempt to locate potential victims by sending probing packets to random targets, exhibit a behavior that is quite different from most legitimate applications. Most prominently, this behavior manifests itself as a large number of (often failed) connection attempts [24, 26].

Other detection techniques based on traffic anomalies check for a large number of connections without previous DNS requests [27] or a large number of received "ICMP unreachable messages" [3]. In addition, there are techniques to identify worms by monitoring traffic sent to dark spaces, which are unused IP address ranges [2], or honeypots [4].

Once malicious traffic flows are identified, a worm has to be contained to prevent further spreading [14]. One technique is based on rate limits for outgoing connections [28]. The idea is that the spread of a worm can be stalled when each host is only allowed to connect to a few new destinations each minute. Another approach is the use of signature-based network intrusion detection systems (such as Snort [18]) that block traffic that contains known worm signatures. Unfortunately, the spreading speed of worms makes it very challenging to load the appropriate signature in a timely manner. To address this problem, techniques have been proposed to automatically extract signatures from network traffic.

The first system to automatically extract signatures from network traffic was Honeycomb [8], which looks for common substrings in traffic sent to a honeypot.

Earlybird [19] and Autograph [6] extend Honeycomb and remove the assumption that all analyzed traffic is malicious. Instead, these systems can identify *recurring byte strings* in general network flows. Our work on polymorphic worm detection is based on these systems. To address the problem of polymorphic worms, which encode themselves differently each time a copy is sent over the network, we propose a novel fingerprinting technique that replaces the string matching with a technique that compares the structural aspects of binary code. This makes the fingerprinting more robust to modifications introduced by polymorphic code and allows us to identify similarities in network flows.

Newsome et al. [15] were the first to point out the problem of string fingerprints in the case of polymorphic worms. Their solution, called Polygraph, proposes capturing multiple invariant byte strings common to all observations of a simulated polymorphic worm. The authors show that certain contiguous byte strings, such as protocol framing strings and the high order bytes of buffer overflow return addresses, usually remain constant across all instances of a polymorphic worm and can therefore be used to generate a worm signature. Our system shares a common goal with Polygraph in that both approaches identify polymorphic worms in network flows. However, we use a different and complementary approach to reach this goal. While Polygraph focuses on multiple invariant byte strings required for a successful exploit, we analyze structural similarities between polymorphic variations of malicious code. This allows our system to detect polymorphic worms that do not contain invariant strings at all. Of course, it is also possible that Polygraph detects worms that our approach misses.

3 Fingerprinting Worms

In this paper, our premise is that at least some parts of a worm contain executable machine code. While it is possible that certain regions of the code are encrypted, others have to be directly executable by the processor of the victim machine (e.g., there will be a decryption routine to decrypt the rest of the worm). Our assumption is justified by the fact that most contemporary worms contain executable regions. For example, in the 2004 “Top 10” list of worms published by anti-virus vendors [21], all entries contain executable code. Note, however, that worms that do not use executable code (e.g., worms written in non-compiled scripting languages) will not be detected by our system.

Based on our assumption, we analyze network flows for the presence of executable code. If a network flow contains no executable code, we discard it immediately. Otherwise, we derive a set of fingerprints for the executable regions. Section 4 provides details on how we identify executable regions and describes the mechanisms we employ to distinguish between likely code and sequences of random data.

When an interesting region with executable code is identified inside a network flow, we generate fingerprints for this region. Our fingerprints are related to the byte strings that are extracted from a network stream by the content sifting approach. To detect polymorphic code, however, we generate fingerprints at a higher level of abstraction that cannot be evaded by simple modifications to the malicious code. In particular, we desire the following properties for our fingerprinting technique:

- **Uniqueness.** Different executable regions should map to different fingerprints. If *identical* fingerprints are derived for *unrelated* executables, the system cannot distinguish between flows that should be correlated (e.g., because they contain variations of the same worm) and those that should not. If the uniqueness property is not fulfilled, the system is prone to producing false positives.
- **Robustness to insertion and deletion.** When code is added to an executable region, either by prepending it, appending it, or interleaving it with the original executable (i.e., *insertion*), the fingerprints for the original executable region should not change. Furthermore, when parts of a region are removed (i.e., *deletion*), the remaining fragment should still be identified as part of the original executable. Robustness against insertion and deletion is necessary to counter straightforward evasion attempts in which an attacker inserts code before or after the actual malicious code fragment.
- **Robustness to modification.** The fingerprinting mechanism has to be robust against certain code modifications. That is, even when a code sequence is modified by operations such as junk insertion, register renaming, code transposition, or instruction substitution, the resulting fingerprint should remain the same. This property is necessary to identify different variations of a single polymorphic worm.

The byte strings generated by the content sifting approach fulfill the uniqueness property, are robust to appending and prepending of padding, and are robust to removal, provided that the result of the deletion operation is at least as long as the analyzed strings. The approach, however, is very sensitive to modifications of the code; even minimal changes can break the byte strings and allow the attacker to evade detection.

Our key observation is that the internal structure of an executable is more characteristic than its representation as a stream of bytes. That is, a representation that takes into account control flow decision points and the sequence in which particular parts of the code are invoked can better capture the nature of an executable and its functionality. Thus, it is more difficult for an attacker to automatically generate variations of an executable that differ in their structure than variations that map to different sequences of bytes.

For our purpose, the structure of an executable is described by its control flow graph (CFG). The nodes of the control flow graph are basic blocks. An edge from a block u to a block v represents a possible flow of control from u to v. A basic block describes a sequence of instructions without any jumps or jump targets in the middle.

Given two regions of executable code that belong to two different network streams, we use their CFGs to determine if these two regions represent two instances of a polymorphic worm. This analysis, however, cannot be based on

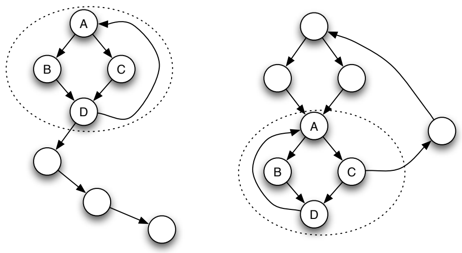


Fig. 1. Two control flow graphs with an example of a common 4-subgraph

simply comparing the entire CFG of the regions because an attacker could trivially evade this technique, e.g., by adding some random code to the end of the worm body before sending a copy. Therefore, we have developed a technique that is capable of *identifying common substructures* of two control flow graphs. We identify common substructures in control flow graphs by checking for isomorphic *connected subgraphs of size k (called k-subgraphs)* contained in all CFGs. Two subgraphs, which contain the same number of vertices k, are said to be isomorphic if they are connected in the same way. When checking whether two subgraphs are isomorphic, we only look at the edges between the nodes under analysis. Thus, incoming and outgoing edges to other nodes are ignored.

Two code regions are *related* if they share common k-subgraphs. Consider the example of the two control flow graphs in Figure 1. While these two graphs appear different at a first glance, closer examination reveals that they share a number of common 4-subgraphs. For example, nodes A to D form connected subgraphs that are isomorphic. Note that the number of the incoming edges is different for the A nodes in both graphs. However, only edges from and to nodes that are part of the subgraph are considered for the isomorphism test.

Different subgraphs have to map to different fingerprints to satisfy the uniqueness property. The approach is robust to insertion and deletion because two CFGs are related as long as they share sufficiently large, isomorphic subgraphs. In addition, while it is quite trivial for an attacker to modify the string representation of an executable to generate many variations automatically, the situation is different for the CFG representation. Register renaming and instruction substitution (assuming that the instruction is not a control flow instruction) have no influence on the CFG. Also, the reordering of instructions within a basic block and the reordering of the layout of basic blocks in the executable result in the same control flow graph. This makes the CFG representation more robust to code modifications in comparison to the content sifting technique. Of course, an adversary can attempt to evade our system by introducing code modifications that change the CFG of the worm. Such and other limitations of our approach are discussed in Section 8.

To refine the specification of the control flow graph, we also take into account information derived from each basic block, or, to be more precise, from the instructions in each block. This allows us to distinguish between blocks that contain significantly different instructions. For example, the system should handle a block that contains a system call invocation differently from one that does not. To represent information about basic blocks, a *color* is assigned to each node in the control flow graph. This color is derived from the instructions in each block. The block coloring technique is used when identifying common substructures, that is, two subgraphs (with k nodes) are isomorphic only if the vertices are connected in the same way *and* the color of each vertex pair matches. Using graph coloring, the characterization of an executable region can be significantly improved. This reduces the amount of graphs that are incorrectly considered related and lowers the false positive rate.

4 Control Flow Graph Extraction

The initial task of our system is to construct a control flow graph from a network stream. This requires two steps. In the first step, we perform a linear disassembly of the byte stream to extract the machine instructions. In the second step, based on this sequence of instructions, we use standard techniques to create a control flow graph.

One problem is that it is not known *a priori* where executable code regions are located within a network stream or whether the stream contains executable code at all. Thus, it is not immediately clear which parts of a stream should be disassembled. The problem is exacerbated by the fact that for many instruction set architectures, and in particular for the Intel x86 instruction set, most bit combinations map to valid instructions. As a result, it is highly probable that even a stream of random bytes could be disassembled into a valid instruction sequence. This makes it very difficult to reliably distinguish between valid code areas and random bytes (or ASCII text) by checking only for the presence or absence of valid instructions.

We address this problem by disassembling the entire byte stream first and deferring the identification of "meaningful" code regions after the construction of the CFG. This approach is motivated by the observation that the structure (i.e., the CFG) of actual code differs significantly from the structure of random instruction sequences. The CFG of actual code contains large clusters of closely connected basic blocks, while the CFG of a random sequence usually contains mostly single, isolated blocks or small clusters. The reason is that the disassembly of non-code byte streams results in a number of invalid basic blocks that can be removed from the CFG, causing it to break into many small fragments. A basic block is considered invalid **(i)** if it contains one or more invalid instructions, **(ii)** if it is on a path to an invalid block, or **(iii)** if it ends in a control transfer instruction that jumps into the middle of another instruction.

As mentioned previously, we analyze connected components with at least k nodes (i.e., k-subgraphs) to identify common subgraphs. Because random instruction sequences usually produce subgraphs that have less than k nodes, the vast majority of non-code regions are automatically excluded from further analysis. Thus, we do not require an explicit and *a priori* division of the network

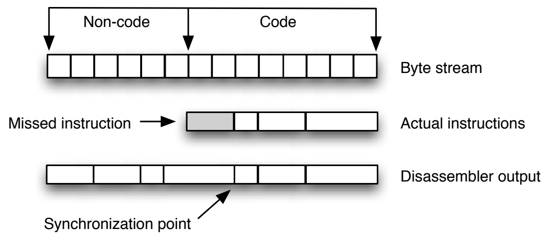


Fig. 2. Linear disassembler misses the start of the first valid instruction

stream into different regions nor an oracle that can determine if a stream contains a worm or not, as is required by the approach described in [15]. In Section 7, we provide experimental data that supports the observation that code and non-code regions can be differentiated based on the shape of the control flows.

Another problem that arises when disassembling a network stream is that there are many different processor types that use completely different formats to encode instructions. In our current system, we focus on executable code for Intel x86 only. This is motivated by the fact that the vast majority of vulnerable machines on the Internet (which are the potential targets for a worm) are equipped with Intel x86 compatible processors.

As we perform linear disassembly from the start (i.e., the first byte) of a stream, it is possible that the start of the first valid instruction in that stream is "missed". As we mentioned before, it is probable that non-code regions can be disassembled. If the last invalid instruction in the non-code region overlaps with the first valid instruction, the sequence of actual, valid instructions in the stream and the output of the disassembler will be different (i.e., de-synchronized). An example of a missed first instruction is presented in Figure 2. In this example, an invalid instruction with a length of three bytes starts one byte before the first valid instruction, which is missed by two bytes.

We cannot expect that network flows contain code that corresponds to a valid executable (e.g., in the ELF or Windows PE format), and, in general, it is not possible, to identify the first valid instruction in a stream. Fortunately, two Intel x86 instruction sequences that start at slightly different addresses (i.e., shifted by a few bytes) synchronize quickly, usually after a few (between one and three) instructions. This phenomenon, called *self-synchronizing disassembly*, is caused by the fact that Intel x86 instructions have a variable length and are usually very short. Therefore, when the linear disassembler starts at an address that does not correspond to a valid instruction, it can be expected to re-synchronize with the sequence of valid instructions very quickly [10]. In the example shown in Figure 2, the synchronization occurs after the first missed instruction (shown in gray). After the synchronization point, both the disassembler output and the actual instruction stream are identical.

Another problem that may affect the disassembly of a network stream is that the stream could contain a malicious binary that is obfuscated with the aim of confusing a linear disassembler [10]. In this case, we would have to replace our linear disassembler component with one that can handle obfuscated binaries (for example, the disassembler that we describe in [9]).

5 K-Subgraphs and Graph Coloring

Given a control flow graph extracted from a network stream, the next task is to generate connected subgraphs of this CFG that have exactly k nodes (k-subgraphs).

The generation of k-subgraphs from the CFG is one of the main contributors to the run-time cost of the analysis. Thus, we are interested in a very efficient algorithm even if this implies that not all subgraphs are constructed. A similar decision was made by the authors in [19], who decided to calculate fingerprints only for a certain subset of all strings. The rationale behind their decision is similar to ours. We assume that the number of subgraphs (or substrings, in their case) that are shared by two worm samples is sufficiently large that at least one is generated by the analysis. The validity of this thesis is confirmed by our experimental detection results, which are presented in Section 7.

To produce k-subgraphs, our subgraph generation algorithm is invoked for each basic block, one after another. The algorithm starts from the selected basic block A and performs a depth-first traversal of the graph. Using this depth-first traversal, a spanning tree is generated. That is, we remove edges from the graph so that there is at most one path from the node A to all the other blocks in the CFG. In practice, the depth-first traversal can be terminated after a depth of k because the size of the subgraph is limited to k nodes. A spanning tree is needed because multiple paths between two nodes lead to the generation of many redundant k-subgraphs in which the same set of nodes is connected via different edges. While it would be possible to detect and remove duplicates later, the overhead to create and test these graphs is very high.

Once the spanning tree is built, we generate all possible k-node subtrees with the selected basic block A as the root node. Note that all identified subgraphs

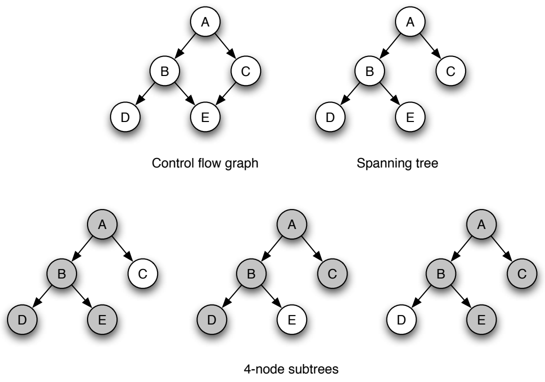


Fig. 3. Example for the operation of the subgraph generation process

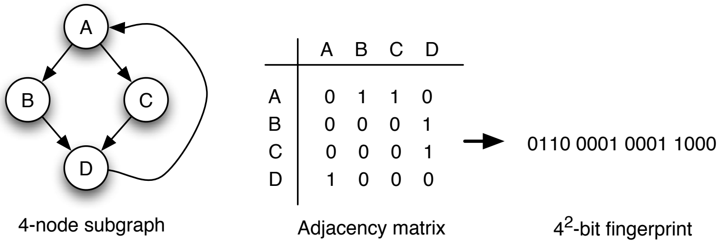
are used in their entirety, also including non-spanning-tree links. Consider the graph shown in Figure 3. In this example, k is 4 and node A is the root node. In the first step, the spanning tree is generated. Then, the subtrees $\{A, B, D, E\}$, $\{A, B, C, D\}$, and $\{A, B, C, E\}$ are identified. The removal of the edge from C to E causes the omission of the redundant subgraph $\{A, B, C, E\}$.

5.1 Graph Fingerprinting

In order to quickly determine which k-subgraphs appear in network streams, it is useful to be able to map each subgraph to a number (a fingerprint) so that two fingerprints are equal only if the corresponding subgraphs are isomorphic. This problem is known as *canonical graph labeling* [1]. The solution to this problem requires that a graph is first transformed into its canonical representation. Then, the graph is associated with a number that uniquely identifies the graph. Since isomorphic graphs are transformed into an identical canonical representation, they will also be assigned the same number.

The problem of finding the canonical form of a graph is as difficult as the graph isomorphism problem. There is no known polynomial algorithm for graph isomorphism testing; nevertheless, the problem has also not been shown to be NP-complete [20]. For many practical cases, however, the graph isomorphism test can be performed efficiently and there exist polynomial solutions. In particular, this is true for small graphs such as the ones that we have to process. We use the `Nauty` library [12, 13], which is generally considered to provide the fastest isomorphism testing routines, to generate the canonical representation of our k-subgraphs. `Nauty` can handle vertex-colored directed graphs and is well suited to our needs.

When the graph is in its canonical form, we use its adjacency matrix to assign a unique number to it. The adjacency matrix of a graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (v_i, v_j) according to whether there is an edge from v_i to v_j or not. As our subgraphs contain a fixed number of vertices k, the size of the adjacency matrix is fixed as well (consisting of k^2 bits). To derive a fingerprint from the adjacency matrix, we simply concatenate its rows and read the result as a single k^2-bit value. This value is unique for each distinct graph since each bit of the fingerprint represents exactly one possible edge. Consider the example in Figure 4 that shows a graph



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 1 & 0 \\ B & 0 & 0 & 0 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 \end{array} \longrightarrow 0110\ 0001\ 0001\ 1000$$

Fig. 4. Deriving a fingerprint from a subgraph with 4 nodes

and its adjacency matrix. By concatenating the rows of the matrix, a single 16-bit fingerprint can be derived.

5.2 Graph Coloring

One limitation of a technique that only uses structural information to identify similarities between executables is that the machine instructions that are contained in basic blocks are completely ignored. The idea of graph coloring addresses this shortcoming.

We devised a graph coloring technique that uses the instructions in a basic block to select a color for the corresponding node in the control flow graph. When using colored nodes, the notion of common substructures has to be extended to take into account color. That is, two subgraphs are considered isomorphic only if the vertices in both graphs are connected in the same way *and* have the same color. Including colors into the fingerprinting process requires that the canonical labeling procedure accounts for nodes of different colors. Fortunately, the `Nauty` routines directly provide the necessary functionality for this task. In addition, the calculation of fingerprints must be extended to account for colors. This is done by first appending the (numerical representation of the) color of a node to its corresponding row in the adjacency matrix. Then, as before, all matrix rows are concatenated to obtain the fingerprint. No further modifications are required to support colored graphs.

It is important that colors provide only a rough indication of the instructions in a basic block; they must not be too fine-grained. Otherwise, an attacker can easily evade detection by producing structurally similar executables with instructions that result in different colorings. For example, if the color of a basic block changes when an `add` instruction is replaced by a semantically equivalent `sub` (subtraction) instruction, the system could be evaded by worms that use simple instruction substitution.

In our current system, we use 14-bit color values. Each bit corresponds to a certain class of instructions. When one or more instructions of a certain class appear in a basic block, the corresponding bit of the basic block's color value is set to 1. If no instruction of a certain class is present, the corresponding bit is 0.

Table 1 lists the 14 color classes that are used in our system. Note that it is no longer possible to substitute an `add` with a `sub` instruction, as both are part of the data transfer instruction class. However, in some cases, it might be possible to replace one instruction by an instruction in another class. For example, the value of register `%eax` can be set to 0 both by a `mov 0, %eax` instruction (which is in the data transfer class) or by a `xor %eax, %eax` instruction (which is a logic instruction). While instruction substitution attacks cannot be completely prevented when using color classes, they are made much more difficult for an attacker. The reason is that there are less possibilities for finding semantically equivalent instructions from different classes. Furthermore, the possible variations in color that can be generated with instructions from different classes is much less than the possible variations on the instruction level. In certain cases,

Table 1. Color classes

Class	Description	Class	Description
Data Transfer	`mov` instructions	String	x86 string operations
Arithmetic	incl. shift and rotate	Flags	access of x86 flag register
Logic	incl. bit/byte operations	LEA	load effective address
Test	test and compare	Float	floating point operations
Stack	push and pop	Syscall	interrupt and system call
Branch	conditional control flow	Jump	unconditional control flow
Call	function invocation	Halt	stop instruction execution

it is even impossible to replace an instruction with a semantically equivalent one (e.g., when invoking a software interrupt).

6 Worm Detection

Our algorithm to detect worms is very similar to the Earlybird approach presented in [19]. In the Earlybird system, the content of each network flow is processed, and all substrings of a certain length are extracted. Each substring is used as an index into a table, called *prevalence table*, that keeps track of how often that particular string has been seen in the past. In addition, for each string entry in the prevalence table, a list of unique source-destination IP address pairs is maintained. This list is searched and updated whenever a new substring is entered. The basic idea is that sorting this table with respect to the substring count and the size of the address lists will produce the set of likely worm traffic samples. That is, frequently occurring substrings that appear in network traffic between many hosts are an indication of worm-related activity. Moreover, these substrings can be used directly as worm signatures.

The key difference between our system and previous work is the mechanism used to index the prevalence table [17]. While Earlybird uses simple substrings, we use the fingerprints that are extracted from control flow graphs. That is, we identify worms by checking for frequently occurring executable regions that have the same structure (i.e., the same fingerprint).

This is accomplished by maintaining a set of network streams S_i for each given fingerprint f_i. Every set S_i contains the distinct source-destination IP address pairs for streams that contained f_i. A fingerprint is identified as corresponding to worm code when the following conditions on S_i are satisfied:

1. m, the number of distinct source-destination pairs contained in S_i, meets or exceeds a predefined threshold M.
2. The number of distinct internal hosts appearing in S_i is at least 2.
3. The number of distinct external hosts appearing in S_i is at least 2.

The last two conditions are required to prevent false positives that would otherwise occur when several clients inside the network download a certain executable file from an external server, or when external clients download a binary from an internal server. In both cases, the traffic patterns are different from the ones generated by a worm, for which one would expect connections between multiple hosts from both the inside and outside networks.

7 Evaluation

7.1 Identifying Code Regions

The first goal of the evaluation of the prototype system was to demonstrate that the system is capable of distinguishing between code and non-code regions of network streams. To accomplish this, the tool was executed over several datasets. The first dataset was composed of the ELF executables from the `/bin` and `/usr/bin` directories of a Gentoo Linux x86 installation. The second dataset was a collection of around 5 Gigabytes of media files (i.e., compressed audio and video files). The third dataset was 1 Gigabyte of random output from OpenBSD 3.6's ARC4 random number generator. The final dataset was a 1.5 Gigabyte selection of texts from the Project Gutenberg electronic book archive. These datasets were selected to reflect the types of data that might commonly be encountered by the tool during the processing of real network traffic. For each of the datasets, the total number of fingerprints, total Kilobytes of data processed, and the number of fingerprints per Kilobyte of data were calculated. For this and all following experiments, we use a value of 10 for k. The results are shown in Table 2.

Table 2. Fingerprint statistics for various datasets

Dataset	Total Fingerprints	Total KB	Fingerprints/KB
Executables	18,882,894	146,750	128.673495
Media	209,348	4,917,802	0.042569
Random	43,267	1,024,000	0.042253
Text	54	1,503,997	0.000036

By comparing the number of fingerprints per Kilobyte of data for each of the datasets, it is clear that the tool can distinguish valid code regions from other types of network data. As asserted in Section 4, disassemblies that contain invalid instruction sequences within basic blocks or a lack of sufficiently connected basic blocks produce many subgraphs with less than 10 nodes. Since a fingerprint is only produced for a subgraph with at least 10 nodes, one expects the rate of fingerprints per Kilobyte of data to be quite small, as we see for the media, random, and text datasets. On the other hand, disassemblies that produce large, strongly-connected graphs (such as those seen from valid executables) result in a large rate of fingerprints per Kilobyte, as we see from the executables dataset.

7.2 Fingerprint Function Behavior

As mentioned in Section 3, the fingerprints generated by the prototype system must ideally be "unique" so that different subgraphs will not map to the same

Table 3. Fingerprint collisions for coreutils dataset

Fingerprints	Total Collisions	Collision Rate	Mismatched Coll.	Mismatch Rate
83,033	17,320	20.86%	84	0.10%

fingerprint. To evaluate the extent to which the system adheres to this property, the following experiment was conducted to determine the rate of fingerprint collisions from non-identical subgraphs. The prototype was first run over a set of 61 ELF executables from the Linux coreutils package that had been compiled with debugging information intact, including the symbol table. The fingerprints and corresponding subgraphs produced during the run were extracted and recorded. An analyzer then processed the subgraphs, correlating each node's address with the symbol table of the corresponding executable to determine the function from which the subgraph was extracted. Finally, for those fingerprints that were produced by subgraphs from multiple executables, the analyzer compared the list of functions the subgraphs had been extracted from. The idea was to determine whether the fingerprint collision was a result of shared code or rather was a violation of the fingerprint uniqueness property. Here, we assume that if all subgraphs were extracted from functions that have the same name, they are the result of the same code. The results of this experiment are shown in Table 3.

From the table, we can see that for the coreutils package, there is a rather large fingerprint collision rate, equal to about 21%. This, however, was an expected result; the coreutils package was chosen as the dataset for this experiment in part because all executables in the package are statically linked with a library containing utility functions, called `libfetish`. Since static linking implies that code sections are copied directly into executables that reference those sections, a high degree of code sharing is present in this dataset, resulting in the observed fingerprint collision rate.

The mismatched collisions column records the number of fingerprint collisions between subgraphs that could not be traced to a common function. In these cases, we must conclude that the fingerprint uniqueness property has been violated, and that two different subgraphs have been fingerprinted to the same value. The number of such collisions in this experiment, however, was very small; the entire run produced a mismatched collision rate of about 0.1%.

As a result of this experiment, we conclude that the prototype system produces fingerprints that generally map to unique subgraphs with an acceptably small collision rate. Additionally, this experiment also demonstrates that the tool can reliably detect common subgraphs resulting from shared code across multiple analysis targets.

7.3 Analysis of False Positive Rates

In order to evaluate the degree to which the system is prone to generating false detections, we evaluated it on a dataset consisting of 35.7 Gigabyte of network traffic collected over 9 days on the local network of the Distributed Systems

Table 4. Incorrectly labeled fingerprints as a function of M. 1,400,174 total fingerprints were encountered in the evaluation set.

M	3	4	5	6	7	8	9	10	11
Fingerprints	12,661	7,841	7,215	3,647	3,441	3,019	2,515	1,219	1,174
M	12	13	14	15	16	17	18	19	20
Fingerprints	1,134	944	623	150	44	43	43	24	23
M	21	22	23	24	25				
Fingerprints	22	22	22	22	22				

Group at the Technical University of Vienna. This evaluation set contained 661,528 total network streams and was verified to be free of known attacks. The data consists to a large extent of HTTP (about 45%) and SMTP (about 35%) traffic. The rest is made up of a wide variety of application traffic including SSH, IMAP, DNS, NTP, FTP, and SMB traffic.

In this section, we explore the degree to which false positives can be mitigated by appropriately selecting the detection parameter M. Recall that M determines the number of unique source-destination pairs that a network stream set S_i must contain before the corresponding fingerprint f_i is considered to belong to a worm. Also recall that we require that a certain fingerprint must occur in network streams between two or more internal and external hosts, respectively, before being considered as a worm candidate. False positives occur when legitimate network usage is identified as worm activity by the system. For example, if a particular fingerprint appears in too many (benign) network flows between multiple sources and destinations, the system will identify the aggregate behavior as a worm attack. While intuitively it can be seen that larger values of M reduce the number false positives, they simultaneously delay the detection of a real worm outbreak.

Table 4 gives the number of fingerprints identified by the system as suspicious for various values of M. For comparison, 1,400,174 total fingerprints were observed in the evaluation set. This experiment indicates that increasing M beyond 20 achieves diminishing returns in the reduction of false positives (for this traffic trace). The remainder of this section discusses the root causes of the false detections for the 23 erroneously labeled fingerprint values for $M = 20$.

The 23 stream sets associated with the false positive fingerprints contained a total of 8,452 HTTP network flows. Closer inspection of these showed that the bulk of the false alarms were the result of binary resources on the site that were (a) frequently accessed by outside users and (b) replicated between two internal web servers. These accounted for 8,325 flows (98.5% of the total) and consisted of:

– 5544 flows (65.6%): An image appearing on most of the pages of a Java programming language tutorial.
– 2148 flows (25.4%): The image of the research group logo, which appears on many local pages.
– 490 flows (5.8%): A single Microsoft PowerPoint presentation.

– 227 flows (2.7%): Multiple PowerPoint presentations that were found to contain common embedded images.

The remaining 43 flows accounted for 0.5% of the total and consisted of external binary files that were accessed by local users and had fingerprints that, by random chance, collided with the 23 flagged fingerprints.

The problem of false positives caused by heavily accessed, locally hosted files could be addressed by creating a *white list* of fingerprints, gathered manually or through the use of an automated web crawler. For example, if we had prepared a white list for the 23 fingerprints that occurred in the small number of image files and the single PowerPoint presentation, we would not have reported a single false positive during the test period of 9 days.

7.4 Detection Capabilities

In this section, we analyze the capabilities of our system to detect polymorphic worms. Polymorphism exists in two flavors. On one hand, an attacker can attempt to camouflage the nature of the malicious code using encryption. In this case, many different worm variations can be generated by encrypting the payload with different keys. However, the attacker has to prepend a decryption routine before the payload. This decryption routine becomes the focus of defense systems that attempt to identify encrypted malware. The other flavor of polymorphism (often referred to as metamorphism) includes techniques that aim to modify the malicious code itself. These techniques include the renaming of registers, the transposition of code blocks, and the substitution of instructions. Of course, both techniques can be combined to disguise the decryption routine of an encrypted worm using metamorphic techniques.

In our first experiment, we analyzed malicious code that was disguised by ADMmutate [11], a well-known polymorphic engine. ADMmutate operates by first encrypting the malicious payload, and then prepending a metamorphic decryption routine. To evaluate our system, we used ADMmutate to generate 100 encrypted instances of a worm, which produced a different decryption routine for

Table 5. Malware variant detection within families

Family	Variant Tests	Matches	Match Rate
FIZZER	1	1	100.00%
FRETHEM	1	1	100.00%
KLEZ	6	6	100.00%
KORGO	136	9	0.07%
LOVGATE	300	300	100.00%
MYWIFE	3	1	0.33%
NIMDA	1	1	100.00%
OPASERV	171	11	0.064%
All	1,991	338	16.97%

each run. Then, we used our system to identify common substructures between these instances.

Our system could not identify a single fingerprint that was common to all 100 instances. However, there were 66 instances that shared one fingerprint, and 31 instances that shared another fingerprint. Only 3 instances did not share a single common fingerprint at all. A closer analysis of the generated encryption routines revealed that the structure was identical between all instances. However, ADMmutate heavily relies on instruction substitution to change the appearance of the decryption routine. In some cases, data transfer instructions were present in a basic block, but not in the corresponding block of other instances. These differences resulted in a different coloring of the nodes of the control flow graphs, leading to the generation of different fingerprints. This experiment brings to attention the possible negative impact of colored nodes on the detection. However, it also demonstrates that the worm would have been detected quickly since a vast majority of worm instances (97 out of 100) contain one of only two different fingerprints.

The aim of our second experiment was to analyze the structural similarities between different members of a worm family. Strictly speaking, members of a worm family are not polymorphic *per se*, but the experiment provides evidence of how much structural similarity is retained between variations of a certain worm. This is important to understand how resilient our system is to a surge of worm variations during an outbreak.

For this experiment, the prototype was run against 342 samples of malware variants from 93 distinct families. The fingerprints generated for each of the malware variants were extracted and recorded. An analyzer then performed a pairwise comparison between each member of each family, searching for common fingerprints. If a common fingerprint was found, a match between the family variants was recorded. Table 5 summarizes some of the more interesting results of this experiment.

From the results, one can see that certain malware variants retain significant structural similarity within their family. Notably, all 25 `LOVGATE` variants share common structural characteristics with one another. There are, however, many cases in which the structural characteristics between variants differs greatly; manual inspection using IDA Pro verified that our system was correct in not reporting common fingerprints as the CFGs were actually very different. While one might consider this disappointing, recall instead that it is rather difficult for an attacker to implement a worm that substantially and repeatedly mutates its structure after each propagation while retaining its intended functionality. Thus, the experiment should demonstrate that the prototype is capable of detecting similarity between real-world examples of malware when it is present.

8 Limitations

One limitation of the current prototype is that it operates off-line. Our experiments were performed on files that were captured from the network and later analyzed. As future work, we plan to implement the necessary infrastructure to operate the system on-line.

Related to this problem is that our analysis is more complex, and, thus, more costly than approaches that are based on substrings [6, 19]. Not only is it necessary to parse the network stream into instructions, we also have to build the control flow graph, generate subgraphs, and perform canonical graph labeling. While many network flows do not contain executables, thus allowing us to abort the analysis process at an early stage, performance improvements are necessary to be able to deploy the system on-line on fast network links. Currently, our system can analyze about 1 Megabyte of data per second. Most of the processing time is spent disassembling the byte stream and generating the CFG.

A key advantage of our approach over the Earlybird [19] and Autograph [6] systems is that our system is more robust to polymorphic modifications of a malicious executable. This is due to the fact that we analyze the structure of an executable instead of its byte stream representation. However, an attacker could attempt to modify the structure of the malicious code to evade detection. While one-time changes to the structure of a binary are quite possible, the automatic generation of semantically equivalent code pieces that do not share common substructures is likely more challenging. Another possibility to erode the similarities between worm instances is to insert conditional branches into the code that are never taken. This can be done at a low cost for the attacker, but it might not be straightforward to generate such conditional branches that cannot be identified by a more advanced static analysis. A possibly more promising attack venue for a worm author is to attack the coloring scheme. By finding instructions from different classes, worm variations can be obtained that are considered different by our system. The experimental results for ADMmutate in the previous section have demonstrated that the system can be forced to calculate different fingerprints for the decryption routine. However, the results have also shown that, despite appearing completely different on a byte string level, the total number of fingerprints is very low. In this case, detection is delayed, but because of the small number of variations, the worm will eventually be automatically identified.

Finally, our technique cannot detect malicious code that consists of less than k blocks. That is, if the executable has a very small footprint we cannot extract sufficient structural information to generate a fingerprint. We chose 10 for k in our experiments, a value that seems reasonable considering that the Slammer worm, which is only 376 bytes long and fits into a single UDP packet, has a CFG with 16 nodes. For comparison, CodeRed is about 4 Kilobytes long and has a CFG with 127 nodes.

9 Conclusions

Worms are automated threats that can compromise a large number of hosts in a very small amount of time, making human-based countermeasures futile. In the past few years, worms have evolved into sophisticated malware that supports optimized identification of potential victims and advanced attack techniques. Polymorphic worms represent the next step in the evolution of this type of malicious software. Such worms change their binary representation as part of the spreading process, making detection and containment techniques based on the identification of common substrings ineffective.

This paper presented a novel technique to reliably identify polymorphic worms. The technique relies on structural analysis and graph coloring techniques to characterize the high-level structure of a worm executable. By abstracting from the concrete implementation of a worm, our technique supports the identification of different mutations of a polymorphic worm.

Our approach has been used as the basis for the implementation of a system that is resilient to a number of code transformation techniques. This system has been evaluated with respect to a large number of benign files and network flows to demonstrate its low rate of false positives. Also, we have provided evidence that the system represents a promising step towards the reliable detection of previously unknown, polymorphic worms.

References

1. L. Babai annd E. Luks. Canonical Labeling of Graphs. In *15th ACM Symposium on Theory of Computing*, 1983.
2. M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
3. V. Berk, R. Gray, and G. Bakos. Using Sensor Networks and Data Fusion for Early Detection. In *SPIE Aerosense Conference*, 2003.
4. D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levin, and Henry O. HoneyStat: Local Worm Detection Using Honeypots. In *7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2004.
5. T. DeTristan, T. Ulenspiegel, Y. Malcom, and M. von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. `http://www.phrack.org/show.php?p=61&a=9`.
6. H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *13th Usenix Security Symposium*, 2004.
7. O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical report, Georgia Tech, 2004.
8. C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *2nd Workshop on Hot Topics in Networks*, 2003.
9. C. Kruegel, F. Valeur, W. Robertson, and G. Vigna. Static Analysis of Obfuscated Binaries. In *13th Usenix Security Symposium*, 2004.
10. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.
11. S. Macaulay. ADMmutate: Polymorphic Shellcode Engine. `http://www.ktwo.ca/ttsecurity.html`.
12. B. McKay. Nauty: No AUTomorphisms, Yes? `http://cs.anu.edu.au~bdm/nauty/`.
13. B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30, 1981.
14. D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *IEEE Infocom Conference*, 2003.

15. J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Symposium on Security and Privacy*, 2005.
16. V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *7th Usenix Security Symposium*, 1998.
17. M. O. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Techonology, Harvard University, 1981.
18. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Usenix LISA Conference*, 1999.
19. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.
20. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory*, chapter Graph Isomorphism. Addison-Wesley, 1990.
21. Sophos. War of the Worms: Top 10 list of worst virus outbreaks in 2004. `http://www.sophos.com/pressoffice/pressrel/uk/20041208yeartopten.html`.
22. S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *2nd ACM Workshop on Rapid Malcode (WORM)*, 2004.
23. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *11th Usenix Security Symposium*, 2002.
24. S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
25. N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *ACM Workshop on Rapid Malcode*, October 2003.
26. N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *13th Usenix Security Symposium*, 2004.
27. D. Whyte, E. Kranakis, and P. van Oorschot. DNS-based Detection of Scanning Worms in an Enterprise Network. In *Network and Distributed Systems Symposium (NDSS)*, 2005.
28. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *18th Annual Computer Security Applications Conference (ACSAC)*, 2002.