1

Behavioral and Structural Properties of Malicious Code

Christopher Kruegel

Secure Systems Lab
Technical University Vienna

Summary. Most current systems to detect malicious code rely on syntactic signatures. More precisely, these systems use a set of byte strings that characterize known malware instances. Unfortunately, this approach is not able to identify previously unknown malicious code for which no signature exists. The problem gets exacerbated when the malware is polymorphic or metamorphic. In this case, different instances of the same malicious code have a different syntactic representation.

In this chapter, we introduce techniques to characterize behavioral and structural properties of binary code. These techniques can be used to generate more abstract, semantically-rich descriptions of malware, and to characterize classes of malicious code instead of specific instances. This makes the specification more robust against modifications of the syntactic layout of the code. Also, in some cases, it allows the detection of novel malware instances.

1.1 Introduction

Malicious code (or malware) is defined as software that fulfills the deliberately harmful intent of an attacker when run. Typical examples of malware include viruses, worms, and spyware. The damage caused by malicious code has dramatically increased in the past few years. This is due to both the popularity of the Internet, which leads to a significant increase in the number of available vulnerable machines, and the sophistication of the malicious code itself.

Current systems to detect malicious code (most prominently, virus scanners) are mostly based on syntactic signatures, which specify byte sequences that are characteristic of particular malware instances. This approach has two drawbacks. First, specifying precise, syntactic signatures makes it necessary to update the signature database whenever a previously unknown malware sample is found. As a result, there is always a window of vulnerability between the appearance of a new malicious code instance and the availability of a signature that can detect it. Second, malicious code can be metamorphic. That is, the malware code mutates while reproducing or spreading across the network, thereby rendering detection using signatures completely ineffective.

In this chapter, we will discuss approaches to characterize higher-level properties of malicious code. These properties are captured by abstract models that describe the behavior and structure of malicious code. The key idea is that semantic or structural properties are more difficult to change between different malware variations. Therefore, our approach results in a more general and robust description of malicious code that is not affected by syntactic changes in the binary image. To demonstrate the effectiveness of our approach, we introduce a technique to describe and detect kernel-level rootkits based on their behavior in Section 1.2. In addition, in Section 1.3, we describe a mechanism to capture the structure of executables and its use to identify metamorphic worms.

1.2 Behavioral Identification of Rootkits

A rootkit is a collection of tools often used by an attacker after gaining administrative privileges on a host. This collection includes tools to hide the presence of the attacker (e.g., log editors), utilities to gather information about the system and its environment (e.g., network sniffers), tools to ensure that the attacker can regain access at a later time (e.g., backdoored servers), and means of attacking other systems. Even though the idea of a rootkit is to provide all the tools that may be needed after a system has been compromised, rootkits focus in particular on backdoored programs and tools to hide the attacker from the system administrator. Originally, rootkits mainly included modified versions of system auditing programs (e.g., `ps` or `netstat` for Unix systems) [10]. These modified programs (also called *Trojan horses*) do not return any information to the administrator about specific files and processes used by the intruder, making the intruder "invisible" to the administrator's eyes. Such tools, however, are easily detected using file integrity checkers such as Tripwire [3].

Recently, a new type of rootkit has emerged. These rootkits are implemented as loadable kernel modules (LKMs). A loadable kernel module is an extension to the operating system (e.g., a device driver) that can be loaded into and unloaded from the kernel at runtime. This runtime kernel extension is supported by many Unix-style operating systems, most notably Solaris and Linux. When loaded, a kernel module has access to the symbols exported by the kernel and can modify any data structure or function pointer that is accessible. Typically, these kernel rootkits "hijack" entries in the system call table and provide modified implementations of the corresponding system call functions [11, 17]. These modified system calls often perform checks on the data passed back to a user process and can thus efficiently hide information about files and processes. An interesting variation is implemented by the `adore-ng` rootkit [18, 19]. In this case, the rootkit does not modify the system call table, but, instead, hijacks the routines used by the Virtual File System (VFS), and, therefore, it is able to intercept (and tamper with) calls that access files in

both the `/proc` file system and the root file system. In any case, once the kernel is infected, it is very hard to determine if a system has been compromised without the help of hardware extensions, such as the TCPA chip [13].

1.2.1 Rootkit Detection

In the following, we introduce a technique for the detection of kernel rootkits in the Linux operating system. The technique is based on the general specification of the behavior of a rootkit. Using static analysis (more precisely, symbolic execution), an unknown kernel module is checked for code that exhibits the malicious behavior. If such code is found, the module is classified as rootkit. The advantage of our method compared to byte string signatures is the fact that our specification describes a general property of a *class* of kernel rootkits. As a result, our technique has the capability to identify previously unknown instances. Also, it is robust to obfuscation techniques that change the syntactic layout of the code but retain its semantics.

The idea for our detection approach is based on the observation that the runtime behavior of regular kernel modules (e.g., device drivers) differs significantly from the behavior of kernel rootkits. We note that regular modules have different goals than rootkits, and thus implement different functionality. Our analysis is performed in two steps. First, we have to specify undesirable behavior. Second, each kernel module binary is statically analyzed for the presence of instructions sequences that implement these specifications.

Currently, our specifications are given informally, and the analysis step has to be adjusted appropriately to deal with new specifications. Although it might be possible to introduce a formal mechanism to model behavioral specifications, it is not necessary for our detection prototype. The reason is that a few general specifications are sufficient to accurately capture the malicious behavior of all current LKM-based rootkits. Nevertheless, the analysis technique is powerful enough that it can be easily extended. This may become necessary when rootkit authors actively attempt to evade detection by changing the code such that it does not adhere to any of our specifications.

1.2.2 Specification of Behavior

A specification of malicious behavior has to model a sequence of instructions that is characteristic for rootkits but that does not appear in regular modules (at least, with a high probability). That is, we have to analyze the behavior of rootkits to derive appropriate specifications that can be used during the analysis step.

In general, kernel modules (e.g., device drivers) initialize their internal data structures during startup and then interact with the kernel via function calls, using both system calls or functions internal to the kernel. In particular, it is not often necessary that a module directly writes to kernel memory. Some exceptions include device drivers that read from and write to memory

areas that are associated with a managed device and that are mapped into the kernel address space to provide more efficient access or modules that overwrite function pointers to register themselves for event callbacks.

Kernel rootkits, on the other hand, usually write directly to kernel memory to alter important system management data structures. The purpose is to intercept the regular control flow of the kernel when system services are requested by a user process. This is done in order to monitor or change the results that are returned by these services to the user process. Because system calls are the most obvious entry point for requesting kernel services, the earliest kernel rootkits modified the system call table accordingly. For example, one of the first actions of the `knark` [11] rootkit is to exchange entries in the system call table with customized functions to hide files and processes.

In newer kernel releases, the system call table is no longer exported by the kernel, and thus it cannot be directly accessed by kernel modules. Therefore, alternative approaches to influence the results of operating system services have been investigated. One such solution is to monitor accesses to the `/proc` file system. This is accomplished by changing the function addresses in the `/proc` file system root node that point to the corresponding read and write functions. Because the `/proc` file system is used by many auditing applications to gather information about the system (e.g., about running processes, or open network connections), a rootkit can easily hide important information by filtering the output that is passed back to the application. An example of this approach is the `adore-ng` rootkit [19] that replaces functions of the virtual file system (VFS) node of the `/proc` file system.

As a general observation, we note that rootkits perform writes to a number of locations in the kernel address space that are usually not touched by regular modules. These writes are necessary either to obtain control over system services (e.g., by changing the system call table, file system functions, or the list of active processes) or to hide the presence of the kernel rootkit itself (e.g., modifying the list of installed modules). Because write operations to operating system management structures are required to implement the needed functionality, and because these writes are unique to kernel rootkits, they present a salient opportunity to specify malicious behavior.

To be more precise, we identify a loadable kernel module as a rootkit based on the following two behavioral specifications:

1. The module contains a data transfer instruction that performs a write operation to an illegal memory area, or
2. the module contains an instruction sequence that i) uses a *forbidden* kernel symbol reference to calculate an address in the kernel's address space and ii) performs a write operation using this address.

Whenever the destination address of a data transfer can be determined statically during the analysis step, it is possible to check whether this address is within a legitimate area. The notion of legitimate areas is defined by a white-list that specifies the kernel addressed that can be safely written to.

For our current system, these areas include function pointers used as event callback hooks (e.g., `br_ioctl_hook()`) or exported arrays (e.g., `blk_dev`).

One drawback of the first specification is the fact that the destination address must be derivable during the static analysis process. Therefore, a complementary specification is introduced that checks for writes to any memory address that is calculated using a forbidden kernel symbol.

A kernel symbol refers to a kernel variable with its corresponding address that is exported by the kernel (e.g., via `/proc/ksysm`). These symbols are needed by the module loader, which loads and inserts modules into the kernel address space. When a kernel module is loaded, all references to external variables that are declared in this module but defined in the kernel (or in other modules) have to be *patched* appropriately. This patching process is performed by substituting the place holder addresses of the declared variables in the module with the actual addresses of the corresponding symbols in the kernel.

The notion of forbidden kernel symbols can be based on black-lists or white-lists. A black-list approach enumerates all forbidden symbols that are likely to be misused by rootkits, for example, the system call table, the root of the `/proc` file system, the list of modules, or the task structure list. A white-list, on the other hand, explicitly defines acceptable kernel symbols that can legitimately be accessed by modules. As usual, a white-list approach is more restrictive, but may lead to false positives when a module references a legitimate but infrequently used kernel symbol that has not been allowed previously. However, following the principle of fail-safe defaults, a white-list also provides greater assurance that the detection process cannot be circumvented.

Note that it is not necessarily malicious when a forbidden kernel symbol is declared by a module. When such a symbol is not used for a *write* access, it is not problematic. Therefore, we cannot reject a module as a rootkit by checking the declared symbols only.

Also, it is not sufficient to check for writes that target a forbidden symbol directly. Often, kernel rootkits use such symbols as a starting point for more complex address calculations. For example, to access an entry in the system call table, the system call table symbol is used as a base address that is increased by a fixed offset. Another example is the module list pointer, which is used to traverse a linked list of module elements to obtain a handle for a specific module. Therefore, a more extensive analysis has to be performed to also track indirect uses of forbidden kernel symbols for write accesses.

Naturally, there is an arms-race between rootkits that use more sophisticated methods to obtain kernel addresses, and our detection system that relies on specifications of malicious behavior. For current rootkits, our basic specifications allow for reliable detection with no false positives (see Section 1.2.4 for details). However, it might be possible to circumvent these specifications. In that case, it is necessary to provide more elaborate descriptions of malicious behavior.

Note that our behavioral specifications have the advantage that they provide a general model of undesirable behavior. That is, these specifications characterize an entire class of malicious actions. This is different from fine-grained specifications that need to be tailored to individual kernel modules.

1.2.3 Symbolic Execution

Based on the specifications introduced in the previous section, the task of the analysis step is to statically check the module binary for instructions that correspond to these specifications. When such instructions are found, the module is labeled as a rootkit.

We perform analysis on binaries using symbolic execution. Symbolic execution is a static analysis technique in which program execution is simulated using symbols, such as variable names, rather than actual values for input data. The program state and outputs are then expressed as mathematical (or logical) expressions involving these symbols. When performing symbolic execution, the program is basically executed with all possible input values simultaneously, thus allowing one to make statements about the program behavior.

In order to simulate the execution of a program, or, in our case, the execution of a loadable kernel module, it is necessary to perform two preprocessing steps.

First, the code sections of the binary have to be disassembled. In this step, the machine instructions have to be extracted and converted into a format that is suitable for symbolic execution. That is, it is not sufficient to simply print out the syntax of instructions, as done by programs such as `objdump`. Instead, the type of the operation and its operands have to be parsed into an internal representation. The disassembly step is complicated by the complexity of the Intel x86 instruction set, which uses a large number of variable-length instructions and many different addressing modes for backward-compatibility reasons.

In the second preprocessing step, it is necessary to adjust address operands in all code sections present. The reason is that a Linux loadable kernel module is merely a standard ELF relocatable object file. Therefore, many memory address operands have not been assigned their final values yet. These memory address operands include targets of jump and call instructions but also source and destination locations of load, store, and move instructions.

For a regular relocatable object file, the addresses are adjusted by the linker. To enable the necessary link operations, a relocatable object also contains, besides regular code and data sections, a set of relocation entries. Note, however, that kernel modules are not linked to the kernel code by a regular linker. Instead, the necessary adjustment (i.e., patching) of addresses takes place during module load time by a special module loader. For Linux kernels up to version 2.4, most of the module loader ran in user-space; for kernels from version 2.5 and up, much of this functionality was moved into the kernel. To

be able to simulate execution, we perform a process similar to linking and substitute place holders in instruction operands and data locations with the real addresses. This has the convenient side-effect that we can mark operands that represent forbidden kernel symbols so that the symbolic execution step can later trace their use in write operations.

When the loadable kernel module has been disassembled and the necessary address modifications have occurred, the symbolic execution process can commence. To be precise, the analysis starts with the kernel module's initialization routine, called `init_module()`. More details about a possible realization of the binary symbolic execution process can be found in [4]. During the analysis, for each data transfer instruction, it is checked whether data is written to kernel memory areas that are not explicitly permitted by the white-list, or whether data is written to addresses that are tainted because of the use of forbidden symbols. When an instruction is found that violates the specification of permitted behavior, the module is flagged as a kernel rootkit.

1.2.4 Evaluation

The proposed rootkit detection algorithm was implemented as a user-space prototype that simulated the object parsing and symbol resolution performed by the existing kernel module loader before disassembling the module and analyzing the code for the presence of malicious writes to kernel memory.

To evaluate the detection capabilities of our system, three sets of kernel modules were created. The first set comprised the `knark` and `adore-ng` rootkits, both of which were used during development of the prototype. As mentioned previously, both rootkits implement different methods of subverting the control flow of the kernel: `knark` overwrites entries in the system call table to redirect various system calls to its own handlers, while `adore-ng` patches itself into the VFS layer of the kernel to intercept accesses to the `/proc` file system. Since each rootkit was extensively analyzed during the prototype development phase, it was expected that all malicious kernel accesses would be discovered by the prototype.

Rootkit	Technique	Description
adore	syscalls	File, directory, process, and socket hiding Rootshell backdoor
all-root	syscalls	Gives all processes UID 0
kbdv3	syscalls	Gives special user UID 0
kkeylogger	syscalls	Logs keystrokes from local and network logins
rkit	syscalls	Gives special user UID 0
shtroj2	syscalls	Execute arbitrary programs as UID 0
synapsys	syscalls	File, directory, process, socket, and module hiding Gives special user UID 0

Table 1.1. Evaluation rootkits.

The second set consisted of a set of seven additional popular rootkits downloaded from the Internet, described in Table 1.1. Since these rootkits were not analyzed during the prototype development phase, the detection rate for this group can be considered a measure of the generality of the detection technique as applied against previously unknown rootkits that utilize similar means to subvert the kernel as `knark` and `adore-ng`.

The final set consisted of a control group of legitimate kernel modules, namely the entire default set of kernel modules for the Fedora Core 1 Linux x86 distribution. This set includes 985 modules implementing various components of the Linux kernel, including networking protocols (e.g., IPv6), bus protocols (e.g., USB), file systems (e.g., EXT3), and device drivers (e.g., network interfaces, video cards). It was assumed that no modules incorporating rootkit functionality were present in this set.

Module Set	Modules Analyzed	Detections	Misclassification Rate
Development rootkits	2	2	0 (0%)
Evaluation rootkits	6	6	0 (0%)
Fedora Core 1 modules	985	0	0 (0%)

Table 1.2. Detection results.

Table 1.2 presents the results of the detection evaluation for each of the three sets of modules. As expected, all malicious writes to kernel memory by both `knark` and `adore-ng` were detected, resulting in a false negative rate of 0% for both rootkits. All malicious writes by each evaluation rootkit were detected as well, resulting in a false negative rate of 0% for this set. We interpret this result as an indication that the detection technique generalizes well to previously unseen rootkits. Finally, no malicious writes were reported by the prototype for the control group, resulting in a false positive rate of 0%. We thus conclude that the detection algorithm is completely successful in distinguishing rootkits exhibiting specified malicious behavior from legitimate kernel modules, as no misclassifications occurred during the entire detection evaluation.

```
kmodscan: initializing scan for rootkits/all-root.o
kmodscan: loading kernel symbol table from boot/System.map
kmodscan: kernel memory configured [c0100000-c041eaf8]
kmodscan: resolving external symbols in section .text
kmodscan: disassembling section .text
kmodscan: performing scan from [.text+40]
kmodscan: WRITE TO KERNEL MEMORY [c0347df0] at [.text+50]
kmodscan: 1 malicious write detected, denying module load
```

Fig. 1.1. `all-root` rootkit analysis.

To verify that the detection algorithm performed correctly on the evaluation rootkits, traces of the analysis performed by the prototype on each rootkit were examined with respect to the corresponding module code. As a simple example, consider the case of the `all-root` rootkit, whose analysis trace is shown in Figure 1.1. From the trace, we can see that one malicious kernel memory write was detected at `.text+50` (i.e., at an offset of 50 bytes into the `.text` section). By examining the disassembly of the `all-root` module, the relevant portion of which is shown in Figure 1.2, we can see that the overwrite occurs in the module's initialization function, `init_module()`[1]. Specifically, the `movl` instruction at `.text+50` is flagged as a malicious write to kernel memory. Correlating the disassembly with the corresponding rootkit source code, shown in Figure 1.3, we can see that this instruction corresponds to the write to the `sys_call_table` array to replace the `getuid()` system call handler with the module's malicious version at line 4. Thus, we conclude that the rootkit's attempt to redirect a system call was properly detected.

```
00000040 <init_module>:
  40:   a1 60 00 00 00          mov    0x60,%eax
  45:   55                      push   %ebp
  46:   89 e5                   mov    %esp,%ebp
  48:   a3 00 00 00 00          mov    %eax,0x0
  4d:   5d                      pop    %ebp
  4e:   31 c0                   xor    %eax,%eax
  50:   c7 05 60 00 00 00 00    movl   $0x0,0x60
  57:   00 00 00
  5a:   c3                      ret
```

Fig. 1.2. `all-root` module disassembly.

```
1 int init_module(void)
2 {
3   orig_getuid = sys_call_table[__NR_getuid];
4   sys_call_table[__NR_getuid] = give_root;
5
6   return 0;
7 }
```

Fig. 1.3. `all-root` initialization function.

[1] Note that this disassembly was generated prior to kernel symbol resolution, thus the displayed read and write accesses are performed on place holder addresses. At runtime and for the symbolic execution, the proper memory address would be patched into the code.

1.3 Structural Identification of Worms

As mentioned previously, polymorphic code can change its binary representation as part of the replication process. This can be achieved by using self-encryption mechanisms or semantics-preserving code manipulation techniques. As a consequence, copies of polymorphic malware often no longer share a common invariant substring that can be used as a detection signature.

In this section, we present a technique that uses the structural properties of an executable to identify different mutations of the same malware. This technique is resilient to code modifications that make existing detection approaches based on syntactic signatures ineffective. Our approach is based on a novel fingerprinting technique based on control flow information that allows us to detect structural similarities between variations of one malware instance or between members of the same malicious code family. The following properties are desirable for the fingerprinting technique:

- **Uniqueness.** Different executable regions should map to different fingerprints. If *identical* fingerprints are derived for *unrelated* executables, the system cannot distinguish between code that should be correlated 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 malware program.

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.[2] Note that a control flow graph is not necessarily a single connected graph. It is possible (and also very likely) that it consists of a number of disjoint components.

Given two regions of executable code that belong to two different malware programs, we use their CFGs to determine if these two regions represent two polymorphic instances of the same code. This analysis, however, cannot be based on 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 actual malware instance. 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 1.4. 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 syntax-based techniques.

[2] More formally, a basic block is defined as a sequence of instructions where the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence. Directed edges between blocks represent jumps in the control flow, which are caused by control transfer instructions (CTIs) such as calls, conditional and unconditional jumps, or return instructions.

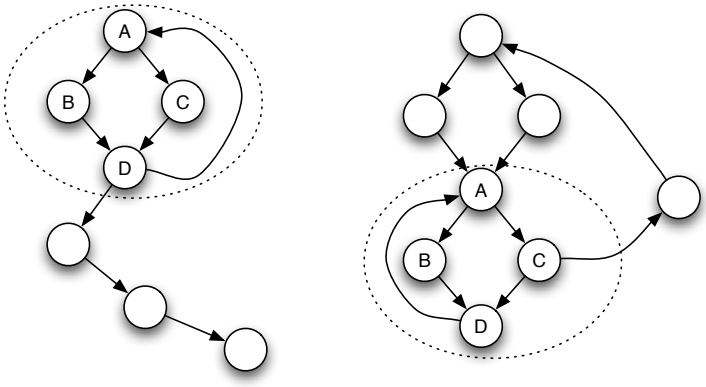

Fig. 1.4. Two control flow graphs with an example of a common 4-subgraph.

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.

1.3.1 Control Flow Graph Extraction

The initial task of our system is to construct a control flow graph from the program(s) that should be analyzed. 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.

Constructing a control flow graph is easy when the executable program is directly available (e.g., as an email attachment or as a file in the file system). However, the situation is very different in the case of network flows. The reason 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. Nevertheless, network traffic must be analyzed to identify

worms. The problem of finding executables in network traffic 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 stream into different regions nor an oracle that can determine if a stream contains a worm or not. Experimental results (presented in [5]) support our claim 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 malware) 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 1.5. 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.

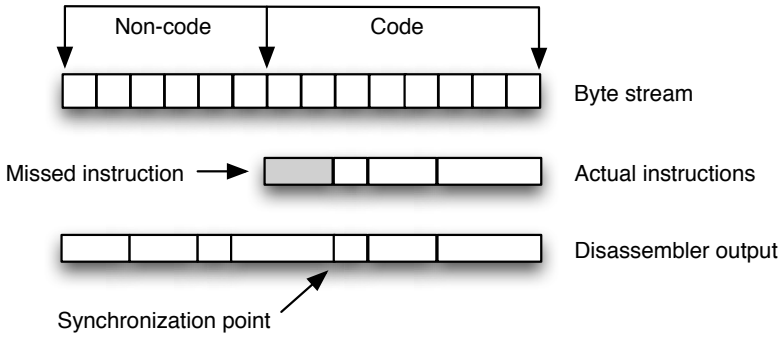

Fig. 1.5. Linear disassembler misses the start of the first valid instruction.

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 [6]. In the example shown in Figure 1.5, 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.

1.3.2 K-Subgraphs and Graph Coloring

Given a control flow graph extracted from a binary program or directly 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. The rationale is that we assume that the number of subgraphs that are shared by two malware 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 1.3.5.

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.

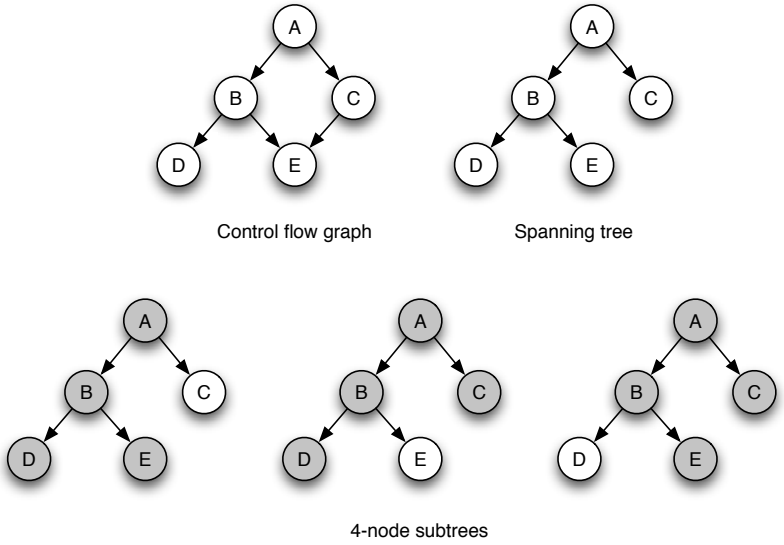

Fig. 1.6. Example for the operation of the subgraph generation process.

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 are used in their entirety, also including non-spanning-tree links. Consider the graph shown in Figure 1.6. 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 $\{A, B, C, E\}$.

1.3.3 Graph fingerprinting

In order to quickly determine which k-subgraphs are shared between different programs or appear in different 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 [15]. 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 [8, 9], 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 1.7 that shows a graph and its adjacency matrix. By concatenating the rows of the matrix, a single 16-bit fingerprint can be derived.

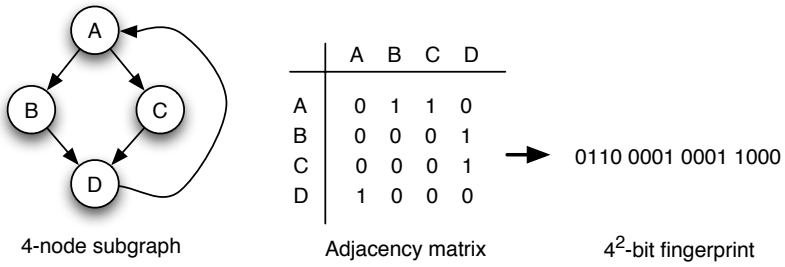

Fig. 1.7. Deriving a fingerprint from a subgraph with 4 nodes.

Of course, when k^2 becomes too large to be practical as a fingerprint, it is also possible to hash the rows of the adjacency matrix instead of concatenating them. In this case, however, fingerprints are no longer unique and a good hash function (for example, one proposed by Jenkins [2]) has to be used to prevent frequent collisions.

1.3.4 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, that is, they must not be too closely associated with specific instructions. 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 malicious code that uses 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.

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

Table 1.3. Color classes.

Table 1.3 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, it is even impossible to replace an instruction with a semantically equivalent one (e.g., when invoking a software interrupt).

1.3.5 Worm Detection

In this section, we show how the previously introduced structural properties of executables can be used to detect polymorphic worms in network traffic. To do so, we have to assume 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 [16], 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.

Our algorithm to detect worms is very similar to the Earlybird approach presented in [14]. 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 [12]. 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.

In a first experiment, we analyzed the capabilities of our system to detect polymorphic worms. To this end, we analyzed malicious code that was disguised by ADMmutate [7], 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 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.

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 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.

We were particularly interested in exploring 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.

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				

Table 1.4. Incorrectly labeled fingerprints as a function of M. 1,400,174 total fingerprints were encountered in the evaluation set.

Table 1.4 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 a 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.

1.4 Conclusions

In this chapter, we have introduced behavioral and structural properties of malicious code. These properties allow a more abstract specification of malware, mitigating shortcomings of syntactic signatures.

Behavioral properties are captured by analyzing the effect of a piece of code on the environment. More precisely, the behavior is specified by checking for the destination addresses of data transfer instructions. In the case of kernel modules, malicious behavior is defined as writes to forbidden regions in the kernel address space. Using symbolic execution, each kernel module is statically analyzed before it is loaded into the kernel. Whenever an illegal write is detected, this module is classified as kernel rootkit and loading is aborted.

The structure of an executable is captured by the subgraphs of the executable's control flow graph. Based on the results of graph isomorphism tests, identical structures that appear in different executables can be identified. The precision of the structural description is further refined by taking into account the classes of instructions (not their exact type) that appear in certain nodes of the control flow graph. Using structural properties of executables, the spread of polymorphic worms can be identified. To this end, our system searches for recurring structures in network flows. When the same structure is identified in connections from multiple source hosts to multiple destinations, this structure is considered to belong to a (possibly polymorphic) worm.

References

1. L. Babai annd E. Luks. Canonical Labeling of Graphs. In *15th ACM Symposium on Theory of Computing*, 1983.
2. R. Jenkins. Hash Functions and Block Ciphers. `http://burtleburtle.net/bob/hash/`.
3. G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. Technical report, Purdue University, November 1993.
4. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *14th Usenix Security Symposium*, 2005.
5. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.
6. 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.
7. S. Macaulay. ADMmutate: Polymorphic Shellcode Engine. `http://www.ktwo.ca/security.html`.
8. B. McKay. Nauty: No AUTomorphisms, Yes? `http://cs.anu.edu.au/~bdm/nauty/`.
9. B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30, 1981.
10. T. Miller. T0rn rootkit analysis. `http://www.ossec.net/rootkits/studies/t0rn.txt`.
11. T. Miller. Analysis of the KNARK Rootkit. `http://www.ossec.net/rootkits/studies/knark.txt`, 2004.
12. M. Rabin. Fingerprinting by Random Polynomials. Technical report, Center for Research in Computing Techonology, Harvard University, 1981.
13. D. Safford. The Need for TCPA. IBM White Paper, October 2002.
14. S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *6th Symposium on Operating System Design and Implementation (OSDI)*, 2004.
15. S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory*, chapter Graph Isomorphism. Addison-Wesley, 1990.
16. Sophos. War of the Worms: Top 10 list of worst virus outbreaks in 2004. `http://www.sophos.com/pressoffice/pressrel/uk/20041208yeartopten.html`.
17. Stealth. adore. `http://spider.scorpions.net/~stealth`, 2001.
18. Stealth. Kernel Rootkit Experiences and the Future. *Phrack Magazine*, 11(61), August 2003.
19. Stealth. adore-ng. `http://stealth.7350.org/rootkits/`, 2004.