Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Catedra de Calculatoare

LUCRARE DE DIPLOMĂ

Analiza aplicaţiilor de tip malware

Conducător Științific:
As.dr.ing. Laura Gheorghe

Autor:
Cristian Condurache

București, 2013

University POLITEHNICA of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

Malware Analysis

Scientific Adviser:
As.dr.ing. Laura Gheorghe

Author:
Cristian Condurache

Bucharest, 2013

Maecenas elementum venenatis dui, sit amet vehicula ipsum molestie vitae. Sed porttitor urna vel ipsum tincidunt venenatis. Aenean adipiscing porttitor nibh a ultricies. Curabitur vehicula semper lacus a rutrum.

Quisque ac feugiat libero. Fusce dui tortor, luctus a convallis sed, lacinia sed ligula. Integer arcu metus, lacinia vitae posuere ut, tempor ut ante.

Abstract

Malware is currently a major security threat for computers and smartphones, with efforts being taken into improving malware detectors with behavior-based detection. In order to classify applications, malware detectors need some form of malicios behavior specification which are usually identified manually by researchers. We present a Linux implementation of the malspec-mining algorithm which automates this process. This algorithm recognizes such specifications by comparing known malicious and benign applications. The output consists of behavior patterns which are specific to the inputted malware and that do not occur in benign applications.

Keywords: behavior-based detection; malspec-mining algorithm; malicious behavior; kernel programming

Contents

List of Figures

List of Tables

Chapter 1

Introduction

From large corporations to the average user, computer and network environment security is an important requirement to which malware is a threat. Malicious software is a program that has been written by an attacker to fulfill a harmful intent. In order to achieve this, the program has to interract with the victim's operating system..

The number of users of a specific operating system is directly correlated to the degree of interest malware writers take in developing software to target that specific operating system. Due to Linux's increasing popularity, better security for operating systems that are based on the Linux kernel has become a necessity. This supports the need for developing tools for Linux malware analysis and improving malware detection methods.

Earlier detection methods focused on analyzing the contents of the executable file of the malware program, such as identifying instruction sequences which were characteristic for specific malware instances. These methods performed poorly when confrunted with unknown malware or new variants of existing ones. Also, in response, attackers started to write malware that modifies its own file while replicating itself, thus eluding these detection methods.

This resulted in a switch to developing behavior based detection systems that are independent from the exact contents of the executable file. Therefore, when analyzing malware samples, analysts started to search for program behavior patterns that suggest a malicious intent. In order for these patterns to work, programs need a higher-level common behavior specification.

The system call interface meets this requirement as malware needs to interract with the operating system to achieve its goals and it is common to all malware. A tipical malware example would be an executable file that replicates itself by reading its own file and then copying it to system directories. This can be captured in a behavior pattern which, compiled into malware specifications, can then be used by malware detectors in order to classify programs based on their behavior.

The project presented in this thesis, named Malsharp, is a Linux tool for automatically searching for malicious program behavior patterns. This tool is a Linux implementation of the malware specification mining algorithm which identifies these behavior patterns by comparing known malware samples to known benign programs. These patterns are a collection of Linux system call parameter dependencies that capture the malicious behavior.

Malsharp is intended to be used by malware specialists to help them analyze new malware samples. It can also be used as part of an automatic detection mechanism which classifies programs based on their behavior by using the malicious behavior specifications.

Chapter 2

State of the Art

Computer security mostly refers to the mechanisms that are used to protect computers and networks from different threats. Although this field deals with many security related issues, one of the major threats to computer security is malware.

2.1 Malware types

Malware, or malicious software, is software programmed and used by attackers in order to gain access to private computers, to obtain sensitive information or to simply disrupt normal computer operation. Malware generically refers to a variety of program forms: viruses, worms, Trojan horses and spyware.

A virus is a program that attempts to replicate itself into other executable files by injecting or replacing code. When the infected programs are run, they can infect other ones in turn. They tipically target common programs which are found on most machines and executables and copy themselves to key directories.

Worms are similar to viruses, only that they spread over the network to other hosts which have the same vulnerability as the initial host. This is done by performing network port scans, DNS queries and then trying to infect other machines. Also, this type of malware usually downloads a second program from a remote server.

Trojan horses, or Trojans, are non-self-replicating code that try to gain privileged access to an operating system and while they seem to be performing a legitimate action they deliver a malicious payload. The payload often contains a backdoor for the attacker that gives him access to the computer or a botnet to send spam or perform Denial-of-service attacks.

Spyware is software created for gathering information without the user's consent. This kind of malware is usually installed without the knowledge of the user or by using deceptive tactics [10].

In Figure 2.1 we can see the procentage of malware samples that correspond to the each type. These statistics were produced by **Panda Security** on April 16$^{\text{th}}$, 2011.

2.2 Avoiding detection

Malware and antimalware software evolution is tightly coupled: when detection methods become more efficient, malware writters use better hiding techniques. In response, anti-malware

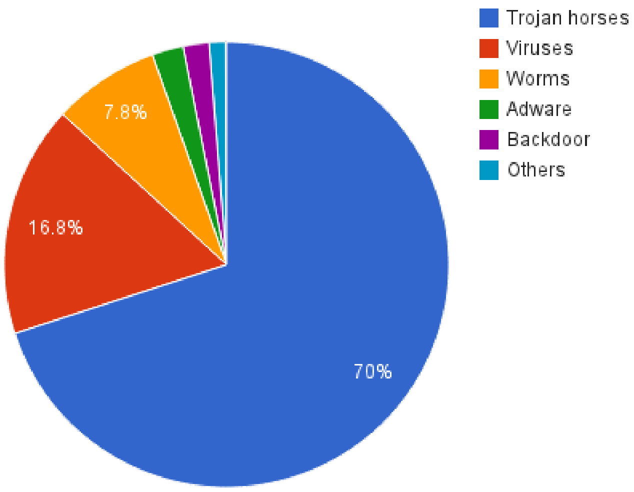

Figure 2.1: Malware Statistics

developers use better algorithms. In the course of time, the development of antimalware software has led attackers to start using different obfuscation methods to avoid detection, such as: concealing API-calling behavior, polymorphic and metamorphic malware.

Malware writers obscure API-calling behavior by using indirect calls. In this respect, API calls can be achieved by using hard coded addresses but this method is incompatible with different versions of the operating system. Another method commonly employed is to define homonymy functions that first locate the API functions addresses and then use the stored address. Also, arrays can be used to store the API function name and the relocation address [6].

Polymorphic viruses use a polymorphic engine to change its executable while keeping the original function intact. A common technique to "morph" viruses is to encrypt the malicious payload and decrypt it at runtime. The encrypted code will then apear to be meaningless and it will be ignored. To obfuscate the decryption routine, the code is transformed by inserting **nop** instructions, permuting register allocation, reordering instructions and inserting jump instructions to maintain the semantics.

Metamorphic viruses also use an engine to change their code in a variety of ways, such as code transposition and substituting instruction sequences with equivalent ones. In addition to this, they interweave their code with the original program's code to trick heuristic detection methods. An important difference between a metamorphic engine and a polymorphic one is that the first can rewrite itself while the second cannot [4].

2.3 Malware detectors

Malware detectors are developed by performing analysis on samples gathered through various means: honeypots, web crawlers, spam traps and security analysts that collect them from infected computers. Bayer *et al.* [3] provided insight into common behavior by analyzing almost one million malware samples by monitoring their network activity and tracking data flows.

Bayer *et al.* created a platform, named Anubis [1], for the dynamic analysis of malware samples which targeted Windows operating systems. The behavior of malware samples was monitored for file system, registry, network and botnet activity, GUI windows and sandbox detection.

Sandboxes are contained environments used to run and test malicious software. The statistics they presented offer insight into common malware behavior and give a hint to what the main goal of malware detectors should be.

Signature based malware detectors use a list of signatures (signature database) to identify known viruses. The signatures are computed by applying a hash function on the malware file. If a part of a program matches a signature entry from the list, then it is classified as malware. This detection method performes very poorly when confrunted with new samples because the signature is unknown. Also, malware writers can easily avoid detection from this type of detectors by using obfuscation techniques in their programs, like polymorphism or metamorphism [5].

Over time, the approach in detecting malware has evolved from analyzing the contents of infected executable files towards identifying malicious or potentially malicious behavior patterns. These patterns are extracted from the malware sample by static or dynamic analysis.

Static analysis of the executable involves scanning the file for particular instruction sequences or different API calls. In order to avoid detection from this type of analysis, attackers attempt to obscure their API-calling behavior or they use a polymorphic engine.

Another method for analysis is to monitor the behavior of the malicious program during runtime in a sandbox, otherwise known as dynamic analysis. This method monitors the malware's interraction with the operating system and the network traffic it produces in order to determine its behavior.

Semantics-aware malware detectors can overcome the problems posed by obfuscation by using specifications of malicious behavior which are not affected by polymorphic malware. By using a higher-level specification, different versions or implementations of malware which perform the same behavior can be detected. Another advantage of this type of detector is that it can also successfully classify unknown malware [8].

2.4 Malspec-Mining Algorithm

The problem with behavior-based detection is that the required specifications have to be manually identified by a malware specialist. The malspec-mining algorithm developed by Christodorescu *et al.* [5] provides a method for automating this otherwise time consuming task.

Their malspec-mining algorithm starts by collecting execution traces from malware and benign programs, then it constructs the corresponding dependence graphs and then it computes the specification of malicious behavior as difference of dependence graphs as minimal contrast subgraph patterns [9].

The malspec-mining algorithm was implemented and tested on a Windows operating system and, although it identified a large number of malware specifications, it managed to capture most of the specifications that were indicated by specialists [5].

In this paper we present an implementation of this algorithm for GNU/Linux based operating systems. In order to capture a program's behavior we developed a system call interceptor and a network traffic interceptor as Linux kernel modules. Then, a user space program reads the traces from the kernel module and constructs a graph where each node represents a system call and the edges represent parameter dependencies. The edges are determined by interpreting the parameter type, direction and value of the recorded system calls. Finally, the malspec-mining algorithm will generate the malicious behavior specifications.

Chapter 3

Malspec Mining Algorithm

3.1 Mining Minimal Contrast Subgraph Patterns

The minimal contrast subgraph patterns used in the malspec mining algorithm were introduced by R. Ming Hieng Ting *et al.* [9]. The following definitions will provide a better understanding of the malspec mining algorithm.

An **edge set** is a labelled graph, *i.e.* a graph with labels attached to its vertices and edges, that has no isolated vertices.

Given two graphs, $\mathrm{G_p}$ and $\mathrm{G_n}$, C is a **common edge set** if and only if C is an edge set and it is a common subgraph. C is a **maximal common common edge set** if it a is a common edge set of the two graphs and if and only if there does not exist a superset which is also a common edge set.

The notions of **maximal common edge set** and **minimal contrast edge sets** are connected and we can determine the second set by obtaining the complement of the first with respect to the original graph and then computing the minimal transversal.

Given G_p and $\{G_{n1}, G_{n2},...,G_{nk}\}$, let M_i be the set of maximal common edge sets between G_p and G_{ni}. Then the set of minimal transversals of $\bar{M_1} \cup \bar{M_2} \cup ... \cup \bar{M_k}$ will be the set of all **minimal contrast edge sets** between G_p and $\{G_{n1}, G_{n2},...,G_{nk}\}$, where $\bar{M_i}$ is the graph complement of M_i with respect to G_p.

For a graph, a **partition** is a set of disjoint and not empty subsets, named cells, of V, *i.e.* the set of all the vertices in the graph. All the vertices in the same cell have the same label and vertices from different cells are labelled differently. The union of all the cells in the partition is equal to V.

Given G_p and G_n that are associated with the partitions T_p and T_n, a **minimal contrast vertex set** is a subset of a cell from T_p such that its cardinality is larger by 1 in comparison with the cells from T_p.

The **minimal contrast subgraph** of a positive graph, G_p, with respect to a negative graph, G_{ni}, is the minimal union of the **minimal contrast edge sets** and the **minimal contrast vertex sets** of the two graphs. The minimal union works like normal union but it removes any graphs that are supergraphs of others in the set.

3.2 Maximal Common Edge Set

In determining the minimal contrast edge set, the most demanding computational task is finding the maximal common edge set. The problem of testing whether a subgraph relationship exists between two graphs is NP-complete.

The maximal common edge set is determined using the backtrack algorithm proposed by J. McGregor for computing the maximal common subgraph [7].

The algorithm takes as input two graphs, G_1 and G_2, with $|V_1| < |V_2|$, and for each node in the G_1 it tries to find a correspondent node from G_2 while maximizing the number of matching edges. The result of the algorithm is a list of pairs of corresponding nodes.

In order to maximize the number of matching edges of the two graphs, a mapping matrix MARCS is used to indicate if an arc from G_1 can correspond to G_2. Initially, all the edges from the first graph can correspond to any edge from the second graph. When a node is paired with another, the edges connected to it can only correspond to edges that are connected to the other node. The edges that do not correspond are marked in MARCS with a zero value.

3.3 Malspec Mining Algorithm

The malspec-mining algorithm collects the execution traces and uses them to create a dependence graph for each program. The following example shows how an execution trace is transformed into a dependence graph.

```
open("/bin/ls",O_RDWR) = 3;
read(3, 0x80000001, 255) = 127;
close(3) = 0;
```

Listing 3.1: System call trace

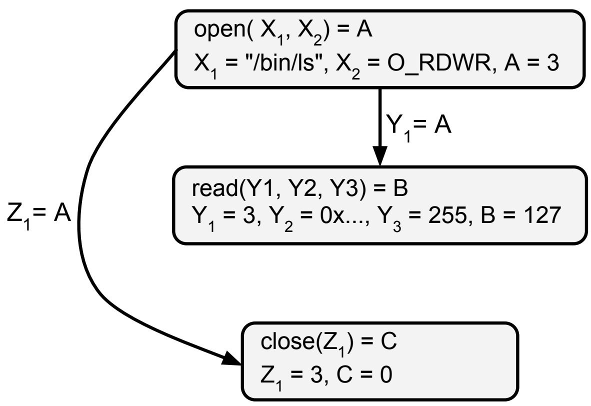

Figure 3.1: System call dependence graph

Each node of the dependence graph represents one system call with its arguments, while de edges represent dependences between arguments of different system calls. The dependence graphs are constructed after aggregating similar operations like multiple reads and writes using the same buffer and file handle, thus resulting in fewer nodes.

Edges are established between nodes which present def-use dependencies: if a previous (in execution order) system call parameter has an out (or inout) parameter with the same value as an in (or inout) argument of a later system call then an edge can be established from the first node to the second node.

Then, the minimal contrast subgraph miner operates in three stages. First, the maximal common edge set for the malware graph and all the benign graphs is determined using the algorithm designed by McGregor [7]. Next, the common edge sets are unioned together and the minimal traversals of their complements are computed, thus obtaining the minimal contrast edge sets for the malware graph. Finally, the contrasts are minimally unioned with the minimal contrast vertex sets to give the complete set of minimal contrast subgraphs.

The resulting subgraphs for each comparison of the malware sample with a benign program are maximally unioned, *i.e.* removing graphs which are subgraphs of others, giving the desired malicious behavior specifications.

3.4 Design

Malsharp takes as input three pathfiles to:

- an XML file which contains the system calls, with argument type and direction, that will be considered for the analysis,
- the malware sample that will be analyzed and
- a file which contains a set of benign programs, one per line.

The architecture of Malsharp that was used in implementing the algorithm is presented in Figure 3.2.

3.4.1 Kernel modules

The kernel modules can be controlled from user space through different **ioctl** commands such as: setting the process id and system calls to be monitored, setting the transport protocol and source and destination port to monitor, reading and removing system call log entries and clearing system call history.

3.4.2 Obtaining traces

The malspec-mining algorithm is run in user space and it will create a new process for each program that it will analyze. Before running the program it will configure the kernel module to monitor the new process and then they will record its execution trace. Only system calls that have entries in the XML input file will be monitored.

The main program will wait for the child process to finish and then it will start reading the execution trace from the system call interceptor driver.

3.4.3 Computing specifications

The malspec-mining algorithm receives the execution trace as input and uses it to create the dependence graph for each program. Because the information gained from monitoring the system calls contains only the register values, the type and direction information for each argument must be filled. This is done by parsing the XML file given as input to Malsharp.

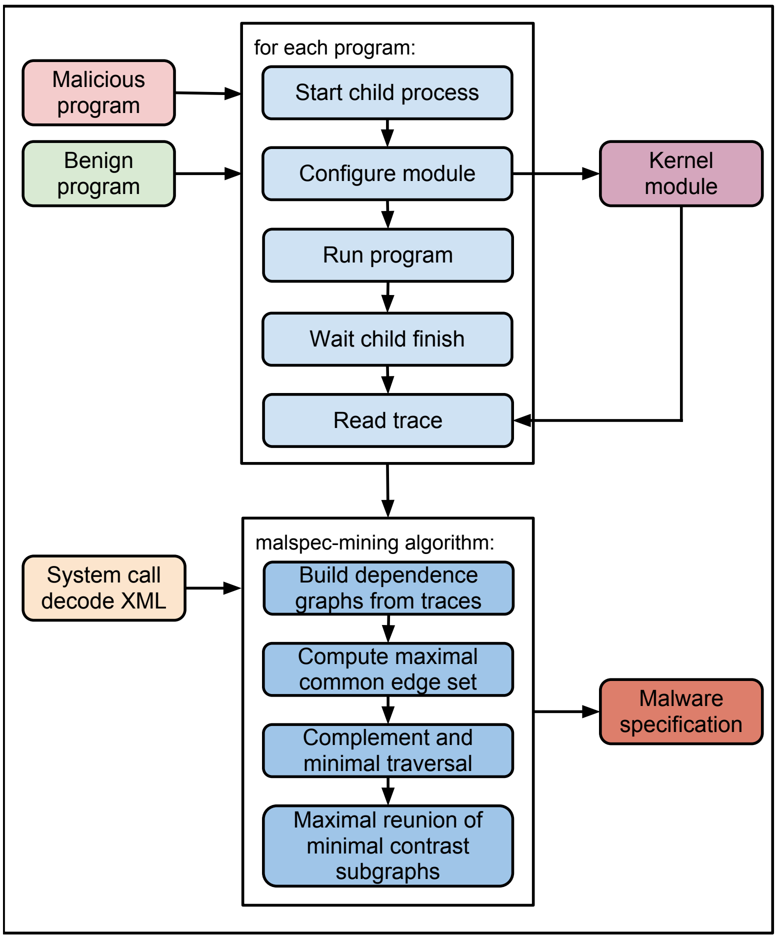

Figure 3.2: Architecture

Next, each pair of graphs is run through the different stages of the algorithm. The output consists of a set of malware specifications that are behavior patterns found in the malware sample that did not occur in any of the benign programs. These specifications can later be used for detecting malware.

Chapter 4

Implementation

Malsharp was implemented in C++, a natural choice considering we had to comunicate with the Linux kernel modules which are implemented in C. Both the kernel module and the userspace application use the same header file that contains the common data structures.

4.1 System Call Interceptor Driver

The System Call Interceptor Driver (SCID) can be configured by giving the pid of the process as an argument at module insertion or by opening the device and calling `ioctl`. The system calls to be monitored can only be configured by `ioctl`.

The SCID registers itself as a character device named **scid** using the misc device interface provided by the Linux kernel. It also creates a symlink to procfs that is used by the userspace part of the application for opening the device.

This driver logs only the system call number and the arguments and return values. The system calls are intercepted by replacing the original value of entry in the system call table with the address of an interceptor function which will run the system call and it will perform the necessary logging.

System call history is implemented with a queue, which contains the recorded system call values and the actual system call number. When an `ioctl` read call is received from userspace, it will remove and return the first entry in the queue. It is also possible to clear the entire log history if needed or to get the total number of entries from the queue.

The kernel module uses the `syscall_params` struct to retrieve the parameters from the stack when intercepting system calls. The data structure returned by the kernel is called `sctrace_t` and it contains the system call number, a syscall_params struct with the argument values and the return value.The following listing contains the definitions for the data structures used by the kernel module.

```
/* data struct for syscall intercepting */
struct syscall_params {
        long ebx, ecx, edx, esi, edi, ebp, eax;
};

/* data struct for read syscall history */
typedef struct _sctrace_t {
        int sc_no;
```

```
        struct syscall_params sc_params;
        long ret;
} sctrace_t;
```

Listing 4.1: SCID data structures

The SCID module `ioctl` commands are defined using the macros provided by the Linux kernel. For the necessary code value required in the kernel macros we chose `0xA1`, which is currently unused according to the documentation file in the kernel.

```
#define IOCTL_SET_PID           _IOW(0xA1, 1, int)
#define IOCTL_ADD_SYSCALL       _IOW(0xA1, 2, int)
#define IOCTL_DEL_SYSCALL       _IOW(0xA1, 3, int)
#define IOCTL_CLEAR_HISTORY     _IO(0xA1, 4)
#define IOCTL_COUNT_HISTORY     _IOR(0xA1, 5, int)
#define IOCTL_READ_HISTORY      _IOR(0xA1, 6, sctrace_t)
```

Listing 4.2: Command macros for ioctl

- `IOCTL_SET_PID` sets the pid of the process that will be monitored
- `IOCTL_ADD_SYSCALL` start monitoring a system call
- `IOCTL_DEL_SYSCALL` stop monitoring a system call
- `IOCTL_CLEAR_HISTORY` deletes all entries with trace information
- `IOCTL_COUNT_HISTORY` returns the number of trace entries
- `IOCTL_READ_HISTORY` get and remove the first trace entry

4.2 Network Interceptor

The Network Interceptor (NI) is also a kernel module that can be configured in a similar manner to the SCID. This module uses kernel netfilter hooks to intercept packets on PRE_ROUTING chain and it logs statistics such as total number of packets and total size transmitted.

NI is controlled through `ioctl` calls which receive a `intercept_info_t` struct containing the protocol, source and destination ports it should monitor. A zero value destination or source port enables monitoring for all the traffic.

```
typedef struct _intercept_info_t {
        /* IPPROTO_*, TCP, UDP, ICMP */
        unsigned char xport_protocol;
        /* used only in TCP and UDP */
        unsigned short int source;
        unsigned short int dest;
} intercept_info_t;
```

Listing 4.3: parameter data structures

Like SCID, the network interceptor also makes a symlink to procfs from which statistics can be read. The information that is collected includes the total number of packages transmitted and the total information size sent for each pair of source and destination ports.

The information gathered by the NI is not used in the determining the malware specifications at the moment, but integration is possible. Currently, the module can be used to verify if a sample malware program tried to use the network to spread itself or to send information.

4.3 Reading traces

The entries recorded by the SCID module do not contain any type and direction information about the system call. Therefore, in order to successfully identify def-use dependences, the type and direction have to be properly set in a `syscall_t` struct, which is defined below, before the graph is constructed.

```
/* param data structure to describe a syscall's parameters */
#define MAX_NUM_PARAM   8

typedef struct param {
        unsigned char type; /* fd, int, unsigned int, char*, void*,
           unsigned short, size_t */
        unsigned char dir;  /* 1 in, 2 out, 3 inout */
        long value;
} param_t;

typedef struct _syscall_t {
        int syscall_no;               /* the system call number */
        param_t param[MAX_NUM_PARAM]; /* last element is the return
           value */
} syscall_t;
```

Listing 4.4: parameter data structures

The `syscall_t` struct contains a param array with a maximum of 8 argument entries. The first 7 entries are used for the register values of the system call and the 8^{th} entry holds the return value. Possible argument directions are in, out and inout, while data types considered are `int`, `unsigned int`, file descriptor, `char*`, `void*`, `unsigned short` and `size_t`.

Malsharp keeps system call decoding information in an XML file which contains for each system call the type and direction information for each register. The file is parsed using the open source library named **pugixml** [2]. We chose this library because it also supports XPath queries, which are used to query information about particular system calls.

Each program is run separately by creating a new process, making it yield the processor by waiting on a named semaphore and then replacing its image with `execve`. The new process has to be forced to yield the processor in order for the parent process to have sufficient time to set the child's process pid as the target for the interceptor modules. Otherwise, a partial system call trace would be obtained instead and the results would be compromised.

After the parent process has configured the kernel module to monitor its child process, it increments the semaphore to enable it to continue execution. After that, the parent process waits for the child process to end and then it starts reading the execution trace.

4.4 Building dependence graphs

The execution trace inputted is used to create a graph for the sample program. Each node of the graph will contain a `syscall_t` struct. On creation, each graph object receives an array of these structs from which to create the graph's nodes.

After the nodes are created, each pair of nodes is verified for def-use dependences and if such dependencies exist then a new edge is added in a vector as a pair of node pointers. To determine if a def-use dependency exists between two nodes, the two `param` arrays from the `syscall_t` structs are compared.

In addition to system call information, each node contains a unique identification number (or index) in order to uniquely identify nodes within the same graph, even if they have the same system call number and argument values.

Although file descriptors are returned as integer values in Linux, they are treated as different data types because integer values are a common argument type for system calls.

Treating file descriptors the same way as integers would result in false def-use dependences. If an `open` system call would return file descriptor number `3` and a `read` call on a different file would try to read `3` characters, then a false def-use dependence edge would be generated between this pair of nodes.

As pointed out by Christodorescu *et al.* in [5], consecutive system calls of the same type which operate on the same resources can be aggregated into a single node. On a file descriptor, one read of N bytes is equivalent to N consecutive reads of 1 byte.

Aggregation is particulary important because it reduces the number of nodes that have to be paired in the backtracking algorithm designed by McGregor for determining the maximal common edge set.

4.5 Malspec algorithm

McGregor's backtrack algorithm was implemented using a recursive version, although the original pseudocode proposed was iterative.

The algorithm assumed that the graphs are not labelled, but in [9] labelling is presented as one of several powerfull prunning methods. Apart from aggregation, label prunning was also used in implementing the backtracking algorithm: only the nodes which shared the same label, the system call number, were tentatively paired.

After the complements of the maximal common edge sets are computed, the minimal transversal of their union must be computed. In our implementation all the edges represent def-use dependencies, so there is no "weight" attached to the edge labels. Therefore, the minimal transversal of the graph can be computed by doing a simple breadth first search and keeping a list of the edges that added a new node to the queue.

Testing for subgraph and supergraph relationships between two graphs was performed by finding the maximal common subgraph and then testing if every node of a graph was matched by another node from the second graph.

Chapter 5

Evaluation

5.1 Malware test environment

5.2 Test scenarios

5.3 Statistics

Chapter 6

Conclusions

Appendix A

Configuring Monitored System Calls

A.1 syscalls.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<syscalls>
        <syscall no="2" name="fork">
                <ebx type="int"    dir="in"/>
        </syscall>
        <syscall no="3" name="read">
                <ebx type="fd"     dir="in"/>
                <ecx type="pvoid"  dir="out"/>
                <edx type="size_t" dir="out"/>
                <ret type="size_t" dir="out" />
        </syscall>
        <syscall no="5" name="open">
                <ebx type="pchar"  dir="in"/>
                <ecx type="int"    dir="in"/>
                <edx type="ushort" dir="in"/>
                <ret type="fd"     dir="out"/>
        </syscall>
        <syscall no="6" name="close">
                <ebx type="fd"     dir="in"/>
                <ret type="void"   dir="none"/>
        </syscall>
        <syscall no="10" name="unlink">
                <ebx type="pchar"  dir="in"/>
        </syscall>
        <syscall no="11" name="execve">
                <ebx type="pchar"  dir="in"/>
                <ecx type="pchar"  dir="in"/>
                <edx type="pchar"  dir="in"/>
        </syscall>
        <syscall no="12" name="chdir">
                <ebx type="pchar"  dir="in"/>
        </syscall>
```

```
        <syscall no="15" name="chmod">
                <ebx type="pchar"  dir="in"/>
                <ecx type="ushort" dir="in"/>
        </syscall>
        <syscall no="16" name="lchown">
                <ebx type="pchar"  dir="in"/>
                <ecx type="ushort" dir="in"/>
                <edx type="ushort" dir="in"/>
        </syscall>
        <syscall no="24" name="getuid">
                <ret type="uint"   dir="out"/>
        </syscall>
        <syscall no="33" name="access">
                <ebx type="pchar"  dir="in"/>
                <ecx type="int"    dir="in"/>
        </syscall>
        <syscall no="37" name="kill">
                <ebx type="uint"   dir="in"/>
                <ecx type="int"    dir="in"/>
        </syscall>
        <syscall no="38" name="rename">
                <ebx type="pchar"  dir="in"/>
                <ecx type="pchar"  dir="in"/>
        </syscall>
        <syscall no="40" name="rmdir">
                <ebx type="pchar"  dir="in"/>
        </syscall>
        <syscall no="49" name="geteuid">
                <ret type="uint"   dir="out"/>
        </syscall>
        <syscall no="108" name="fstat">
                <ebx type="fd"     dir="in"/>
                <ecx type="pvoid"  dir="inout"/>
        </syscall>
        <syscall no="190" name="vfork">
                <ret type="void"   dir="none"/>
        </syscall>
        <syscall no="199" name="getuid32">
                <ret type="uint"   dir="out"/>
        </syscall>
</syscalls>
```

Listing A.1: System Calls Decode Information XML (syscalls.xml)

Bibliography

[1] Anubis. http://anubis.iseclab.org, 2007.

[2] pugixml, light-weight c++ xml library. http://pugixml.org, 2012.

[3] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. LEET'09 Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms and more, April 2009.

[4] M. Christodorescu and S. Jha. Testing malware detectors. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, July 2004.

[5] M. Christodorescu, S. Jha, and C.Kruegel. Mining specifications of malicious behavior. ESEC-FSE'07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, September 2007.

[6] W. Fu, J. Pang, R. Zhao, Y. Zhang, and B. Wei. Static detection of api-calling behavior from malicious binary executables. International Conference of Computer and Electrical Engineering, December 2008.

[7] James J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. Software - Oractice and Experience, vol. 12, 23-34, 1982.

[8] M.D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics based approach to malware detection. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 2007.

[9] R. Ming Hieng Ting and J. Bailey. Mining minimal contrast subgraph patterns. 6th SIAM international conference on Data Mining, 2006.

[10] M.F. Zolkipli and A. Jantan. Malware behavior analysis: Learning and understanding current malware threats. Second International Conference on Network Applications, Protocols and Services, September 2010.