Study of Common Sub-graphs of System Call Dependency Graphs for Malware Classification

Dylan Marinho

Study of Common Sub-graphs of System Call Dependency Graphs for Malware Classification

Dylan Marinho

Inria Rennes

August 23, 2018

Abstract

Distinguishing legitimate software from malicious software is a problem that requires a lot of expertise. In order to create a malware detection software, an approach consists in extracting *System Call Dependency Graphs* (SCDG) which summarize the behavior of a software. Once SCDGs are extracted, a learning phase identifies sub-graphs characteristic of malicious behaviors. In order to classify the graph of an unknown binary, we look whether it contains such sub-graphs. These techniques proved to be efficient, but no analysis of the sub-graphs extracted during the learning phase has been conducted so far. We study the sub-graphs we find and showcase preprocessing steps on the graphs in order to improve the learning and classification performance. The approach has been applied on graphs extracted from *Mirai*. Mirai is a malware which created a large botnet in 2016 to perform distributed deny of service attacks. We show that the preprocessing step tremendously improve the speed of the learning and classification.

Keywords: Security, *Mirai*, Detection, Malware, SCDG

Reference Internship done from 05/28/2018 to 07/27/2018 in TAMIS, Inria, France under the supervision of Jean Quilbeuf and Axel Legay, Inria.

Contents

1 Introduction

The amount of malware detection by anti-virus companies is exponentially growing [5]. In the recent years, malware attacks have paralyzed transportation systems or hospitals and thus represent a threat for society. Therefore, we need to be able to detect malware as accurately and fast as possible. Currently, human expertise is still required to perform this detection but several techniques exist for automatically detecting malwares. [1].

Syntactic analysis. Malware signatures can be extracted from binaries using a syntactic approach. It consists of analyzing the binary without having to execute it but for instance by scanning it to find specific strings or sequences of bytes. It is the fastest detection technique among the ones presented here. However, some obfuscation techniques modify the binary code while keeping the same behavior, so that the syntactic signature is not present anymore.

Dynamic analysis. In order to be able to analyze the behavior of a program and to circumvent obfuscation techniques changing the syntactic properties of the malware, a common technique is *dynamic analysis*. It consists in executing the malware and observing its interaction with the system. This technique is heavier than syntactic analysis as it requires a concealed environment, called *sandbox* in order to execute the binary.

Concolic analysis. The dynamic analysis is limited because it analyzes only one of the possible execution paths. To address this limitation, *concolic analysis* (a portmanteau of *conc*rete-symb*olic*) has been introduced to extract the behavior of a binary and covering as much of its possible execution paths as possible. It maintains a symbolic representation of the constraints found while analyzing the program and uses a solver to determine whether the studied path is actually reachable. However malicious techniques exist to complicate the conditional constraints, increasing the size of the symbolic representation.

Both dynamic and concolic analysis allow for the inspection of the binary's behavior. A technique to summarize the behavior of a binary is to build a so-called *System Call Dependency Graph* (SCDG) [2, 4]. In order to detect malwares from goodwares, characteristic sub-graphs are extracted from known malware (learning phase) and researched in the SCDGs of the binaries to analyze (detection phase). If a malicious sub-graph is found in the SCDG of a binary to analyze, the latter is labeled as a malware.

The aforementioned approaches have proven to be efficient [4, 8]. However no study of the sub-graph identified as malicious has been conducted so far to our knowledge. In particular, we don't know to which extent they capture the actual malicious part of the behavior.

Therefore, our first contribution is the analysis of the sub-graphs resulting from the learning phase. Although we were not able to give a precise meaning to a given sub-graph, our analysis showed us that many spurious edges originating from the SCDG construction are present in the found sub-graphs. Our research question in this report is thus to understand whether removing these edges, and thus reducing the size of the graphs has an impact on the accuracy and performance of the transformation. Indeed, the number of edges in a collection of graphs influences the time needed to mine common sub-graphs of that collection.

We show in this paper that we can remove spurious edges and propose two methods to do so, which we implement as graph preprocessing step. Furthermore, we evaluate these two

methods on the graphs produced in [8]. Our experimental results show that our first method is not as good as keeping the original graphs, although it reduces the learning time. The second method yields a huge speedup 1,414x for learning and 123x for classification time, while losing only 0.3 points on the $F_{0.5}$ score .

Contributions. This report provides the following contributions.

- Two methods to reduce the size of SCDG, firstly by studying the consistency of their edges and secondly by defining a direction for the system call arguments.
- An experimental evaluation of these methods by experiments: they increase the performances and keep a correct detection rate.

This report is organized in four parts. Section 2 describes the domain, including the construction of SCDG. Section 3 addresses the definition about edges consistency. Section 4 deals with the work on the system call arguments. Section 5 presents the experimental evaluation of the methods and section 6 concludes this report.

2 Background

In this section we present an existing approach for performing behavior-based malware detection. To this end, we first present in Section 2.1 the construction of an SCDG. We then present in Section 2.2 an essential component of the learning and classification of SCDG, namely the common sub-graph mining. Finally, we show in Section 2.3 how the learning and classification is performed on SCDGs , in order to automatically classify the corresponding binaries.

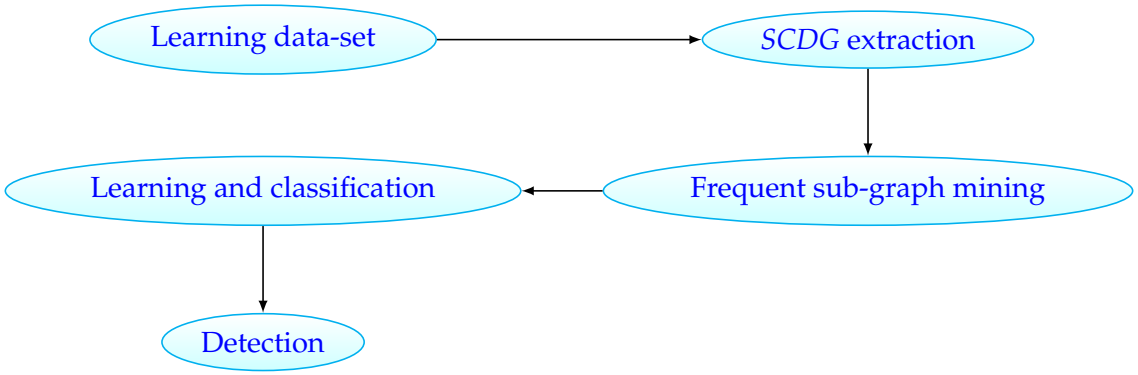

Figure 1: General view of *SCDG* based detection

2.1 SCDG construction

This section presents the construction of SCDGs. We motivate and describe the construction of a *System Call Dependency Graph* (SCDG) from a binary.

2.1.1 Detection by system call dependency

When a programmer writes and builds his software, he has great freedom to manage his functions and arguments (for example, see [8]). In particular, it is possible and sometimes legitimate to produce binaries whose code is obfuscated, by which we mean it is harder to understand. Such techniques include packing [7], that will produce a binary whose only "readable" code is the path that decompresses or decrypts the actual malicious code at runtime. However, the code must comply with standard libraries to communicate and exchange information with the system. In particular, when calling a function from an external library, the binary must respect the calling convention for that function. By capturing the calls to these external function made by the binary, we actually capture all the actions performed on the system and thus the behavior of the program.

Representing the behavior of a binary through its external calls has the following advantages:

- the representation is compact, since only external call are listened;
- it captures all interactions of the binary with the system;
- it is robust to mutations: even if the code is modified or obfuscated, the external calls will remain the same.

Furthermore, capturing the data dependencies between the external calls (i.e. the fact that a given call relies on the information produced by another call) provide an even more robust representation. Indeed, these dependencies are likely to be kept even if the binary is obfuscated. For instance, opening and reading a file, will require a call to `open` that will produce a file handler, which will be passed to a call to `read`. In that example the fact that `read` depends on the call to `open` will be kept event if the code is obfuscated.

2.1.2 Behavioral graphs

We characterize the behavior of a binary as a labeled graph. Here we explain how it is constructed and give an example. We start by formally defining a labeled graph.

Definition 1 (Labeled graph)**.** *A* labeled graph *is a 4-tuple,* $G = (V, E, L, l)$*, where:*

- V *is a set of vertices;*
- $E \subseteq V \times V$ *is a set of edges;*
- L *is a set of labels;*
- $l : V \cup E \to L$ *a function assigning labels to the vertices and edges.*

In order to represent the external calls of the binary and their data dependencies, each binary is transformed into a graph. To do this transformation, the binary is executed (symbolically or dynamically) and a trace of the external calls made by the binary is recorded, along with their arguments and return values. The graph is then constructed from the trace. Each node of the graph correspond to a call made by the software, its label is the name of the call. Then, we add an edge from the node $n1$ to the node $n2$ corresponding to two calls $c1$ and $c2$ if:

- $c1$ occurs before $c2$;
- the value of one of the arguments or the return value of $c1$ is used as an argument of $c2$.

The label of the edge between $n1$ and $n2$ is $a \rightarrow b$ where a is the index of the argument at hand of $c1$ (0 if it is the return value) and b the one of $c2$. Having a label $\ell = a \rightarrow b$, we denote by $src(\ell)$ the index a of the source argument and by $dest(\ell)$ the index b of the destination argument. Such a graph is called a *System Call Dependency Graph* (SCDG).

Consider the code of the Figure 2, which opens the files file.txt and file2.txt, reads 1024 bytes from file, writes them in file2 and closes the files. The graph presented next in Figure 3 is its *SCDG*. In this graph, the nodes of system calls linked with file.txt are colored in yellow and those linked with file2.txt in blue. For example, the open on line 3 creates a variable f which is used by the read on line 5. The two system calls are linked because they share a same value: the graph has an edge between a node open and a node read with a label 0 (return value) $\rightarrow$1 (first argument).

```
1  const char *pathname = 'file.txt';
2  const char *pathname2 = 'file2.txt';
3  int f = open(*pathname,flags);
4  int f2 = open(*pathname2,flags2);
5  ssize_t n = read(f,buf,1024);
6  ssize_t p = write(f2,buf,1024);
7  close(f);
8  close(f2);
```

Figure 2: SCDG example code

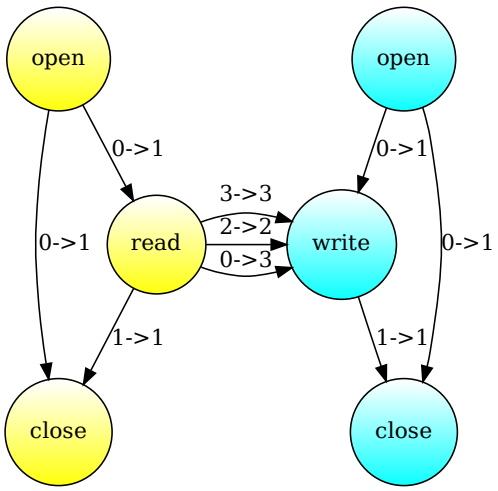

Figure 3: SCDG example graph

2.2 Common sub-graphs extraction

An essential component for performing learning and classification on the SCDG is the extraction of common sub-graphs. We define here formally what a common sub-graph is and present the gSpan algorithm for finding such sub-graphs.

2.2.1 Definition

We start by formally defining what is common sub-graph mining. It consists of finding the sub-graphs of a collection that occur in a large enough subset of it. The *support* specifies a ratio for deciding which sub-graphs are frequent enough: a given sub-graph is frequent enough if the ratio of the number of graphs containing it over the size of the collection is above the support.

Definition 2 (Common Sub-graph Mining)**.** *Given a graph data-set* $GS = (G_i | i \in [\![0, n]\!])$*, a graph* g *and a support* $s \in [0, 1]$*, let*

$$\zeta(g, G) = \begin{cases} 1 & \text{if } g \text{ is a subgraph of } G \\ 0 & \text{if } g \text{ is not a subgraph of } G \end{cases}$$

and

$$\sigma(g, GS) = \sum_{G_i \in GS} \zeta(g, G_i)$$

So, $\overline{\sigma}(g, GS) = \frac{\sigma(g,GS)}{|GS|}$ *denotes the occurrence frequency of* g *in* GS*.*

Frequent sub-graph mining is to find every graph g *such that* $\overline{\sigma}(g, GS)$ *is greater than or equal to* s*.*

2.2.2 gSpan algorithm

gSpan (*Graph-Based Substructure Pattern Mining*) is an algorithm developed by Yan and Han [11] to discovers frequent sub-graphs in graph collection. It builds a lexicographic order on graphs by associating them with a depth-first search (*DFS*). It allows to find all common sub-graphs from a collection of graphs with a certain minimum frequency.

Let $\{A, B, C, \ldots\}$ be a label set for vertices and $\{a, b, c, \ldots\}$ for edges. The algorithm will first discover all the frequent sub-graphs containing an edge A $\xrightarrow{a}$ A. Secondly, it will discover all the frequent sub-graphs containing A $\xrightarrow{a}$ B but not any A $\xrightarrow{a}$ A. This is repeated until the algorithm finds all frequent sub-graphs.

2.3 Learning and classification

We rely on learning to automatically classify unknown suspicious sample. We assume that we have a representative set of binary samples for which the families have been identified (i.e. by experts). We also have a set of binaries to classify. We describe hereafter the learning and classification process.

Learning. For each family F of malware, the learning part have two steps:

1. the extraction of the *SCDGs* for each malware;
2. the extraction of the frequent sub-graphs in that family.

Note that the cleanwares are not used during the learning phase because we don't expect to find common sub-graphs characteristic of goodware behavior. Indeed, goodware can vary a lot, whereas a single malware family is expected to show a lot of commonality between its various samples.

We note $\mathcal{G}_F$ the set of the frequent sub-graphs obtained for the family F and $\mathcal{G}$ their union for all the families of learning.

Classification. The classification is parameterized by a threshold t and has the following steps for each binary b:

1. we extract its *SCDG*, denoted by G;
2. for each sub-graphs $sg \in \mathcal{G}$, we launch gspan(G,sg) to obtain the common sub-graph between G and sg;
3. we compute the amount of sg that is included in b:

$$\mathcal{M}(sg, b) = \frac{\text{size}(\text{gspan}(b, sg))}{\text{size}(sg)}$$

   where $\text{size}(g)$ is the number of edges of the graph g;

4. - if no sub-graph sg is included at more that t (ie. $\forall sg \in \mathcal{G}, \mathcal{M}(sg, b) < t$) we classify it as cleanware;
   - else (ie. $\exists sg \in \mathcal{G}, \mathcal{M}(sg, b) > t$) we classify it as malware. In this case, it is classified as the family with which it obtains the highest score.

2.4 Limits

It is not clear whether all of the edges introduced by the *SCDG* construction from Section 2.1 are useful. Indeed, a small program can generate a big behavioral graph in which some edges do not add information.

***SCDG* size.** Let's consider a program similar to the Figure 2, with an additional call to read and two write. The program in Figure 4 achieves the same as the program in Figure 2 by two read write sequences of 512 bytes instead of a single read write sequence of 1024 bytes. The graph obtained in Figure 5 contains a great number of edges. In order to have a readable graph, all the edges are not drawn:

- a bold edge symbolizes three edges with labels 2−>2, 3−>3, 0−>3;
- a (bold) dashed edge symbolizes four edges with labels 1−>1, 2−>2, 3−>3, 0−>3.

```
1   const char *pathname = 'file.txt';
2   const char *pathname2 = 'file2.txt';
3   int f = open(*pathname,flags);
4   int f2 = open(*pathname2,flags2);
5   ssize_t n = read(f,buf,512);
6   ssize_t p = write(f2,buf,512);
7   ssize_t n2 = read(f,buf,512);
8   ssize_t p2 = write(f2,buf,512);
9   close(f);
10  close(f2);
```

Figure 4: SCDG example code

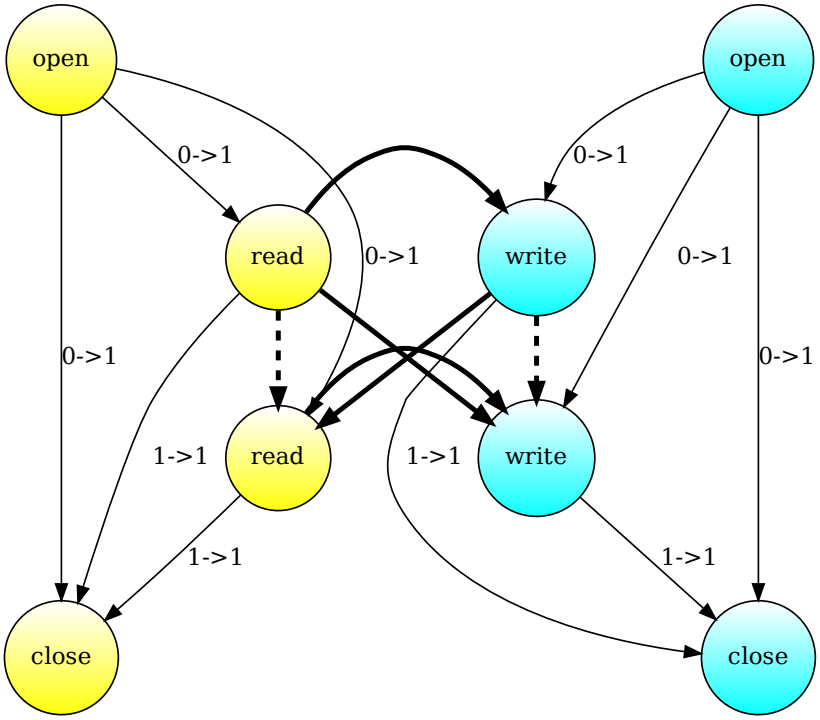

Figure 5: SCDG example graph

Moreover, the number of edges for n cycles of read write is $E(n) = 7n^2 + 2$ (proof in Appendix B). We hope to have a linear expression.

The information contained in the graph from Figure 5 is actually the same as the graph from Figure 7 which contains less edges. Thus adding more edges does not add more information.

***SCDG* consistency** Let's consider the pseudo-code presented in figure 6. We found edges open $\xrightarrow{0\to1}$ socketcall indicating that the return value of open is used as first argument of socketcall. This is possible because the return value of open is the smallest file descriptor available (usually starts as 3 because 0, 1 and 2 are stdin, stdout and stderr) and the first argument of socketcall

```
1   f = open(...)
2   ...
3   s = socketcall(3, ...)
```

Figure 6: Pseudo code

is call which can take values ranging from 1 to 18 [10], 3 in this example.

gSpan complexity. The graph mining being a NP-complete problem, the size (considered as the number of edges) of graphs have a great impact on the process complexity. Thus, keeping the graph size as small as possible while keeping the same amount of information could improve the learning and classification process. We present in the next Sections two preprocessing steps for reducing the size of the produced SCDGs while keeping enough information to ensure an accurate classification.

3 Edges consistency

By studying some sub-graphs extracted from *Mirai* samples [8], it appears that a certain number of edges are caused by artifacts of the execution of the program. For example, the small integers, are often used to encode the action of the system call (parameter call of sys_socketcall, request of ioctl, etc.) or and file descriptors which use the smaller integer available by definition (see Section 2.4 and Appendix A).

Therefore, our algorithm for constructing SCDG, as presented in Section 2.1, will add edges as soon as one of these small integers appears in two different calls. In some cases, such edges are introduced because the value is same but not indicate an actual flow of information and therefore can be removed.

3.1 Consistency definition

From this observation, we formalize the notion of consistency based on the notion of edges type.

Definition 3 (Edge type)**.** *Let $G = (V, E, L, l)$ a labeled graph. We define ET the set of its edges types as:*

$$ET = \{ (l(u), l(v), l((u,v))) \mid (u,v) \in E \}$$

Example 1. *The ET set of the graph defined in figure 3 is:*

$$\{ (open, close, 0\text{->}1), (open, read, 0\text{->}1), (open, write, 0\text{->}1), (read, close, 1\text{->}1),$$
$$(write, close, 1\text{->}1), (read, write, 0\text{->}3), (read, write, 2\text{->}2), (read, write, 3\text{->}3) \}$$

We associate to each edge type present in the *SCDG* to preprocess an interpretation between consistent, meaningless and indeterminate. The first one characterizes edges where the value of the arguments represents the same type of object (eg. fd, first argument of read and write). The second one classifies the edges where the value does not represent the same type of objects. The third case is used because we can't decide based on the *SCDG* only: we would need the

actual value of an argument (eg. for prctl has an argument option which describes its action ; therefore, the information given by arg2 depends on this option value).

Definition 4 (Edge consistency)**.** *A* edge consistency *is a function*

$$EC : ET \longrightarrow \{ \textit{consistent, meaningless, indeterminate} \}$$

Definition 5 (Consistency preprocessing)**.** *Given a graph $\mathcal{G}$, and an edge consistency EC, we define the consistency preprocessing PP_{EC} as (V', E', L', l') such that*

$$\begin{aligned} V' &= V, \\ L' &= L, \\ E' &= \{ (u,v) \in E \mid EC(l(u), l(v), l((u,v))) \in \{ \textit{consistent, indeterminate} \} \}, \\ l' &= l \textit{ restricted to } E' : l\big|_{E'} \end{aligned}$$

In this report we leave the automatic generation of the *EC* function for future work. In our case, the relatively small amount of edge types made it possible to build it manually.

3.2 Limits

As detailed in Section 5.2.1, the number of edge types for the Mirais is 190. This number allows manual classification of the edges as consistent, indeterminate or meaningless.

However, the list of edge types while considering cleanwares and mirais consists of 1,233 elements. With that number of edge types, we need to come up with an automatic way of classifying these edges into the above categories. Moreover, switching to a new data-set would require rebuilding the *EC* function as well.

4 Considering Arguments Direction

Let take back the example given in the figure 5. The ideal graph, which contains all information needed but less edges, can be the one presented in the figure 7. In this graph, we learn that two files are opened, the first is read twice, the second written twice and the finished by being closed. Moreover, the couples read write share the same buffer. In this case, only eight edges are sufficient to capture the dependencies induced by the behavior of the program presented in the Figure 4.

From this example, we try to distinguish the meaning direction of a argument. For example, in the code presented in Figure 2, *buf is an input of write but an output of read. Indeed, the information given by *buf is used by write to modify the file, but it is written by read to be accessed after.

4.1 Definitions

We consider each argument of each system call and we want to associate to it a direction indicating whether the argument is read or written by the call. To this end, we start by formally defining an argument of a system call.

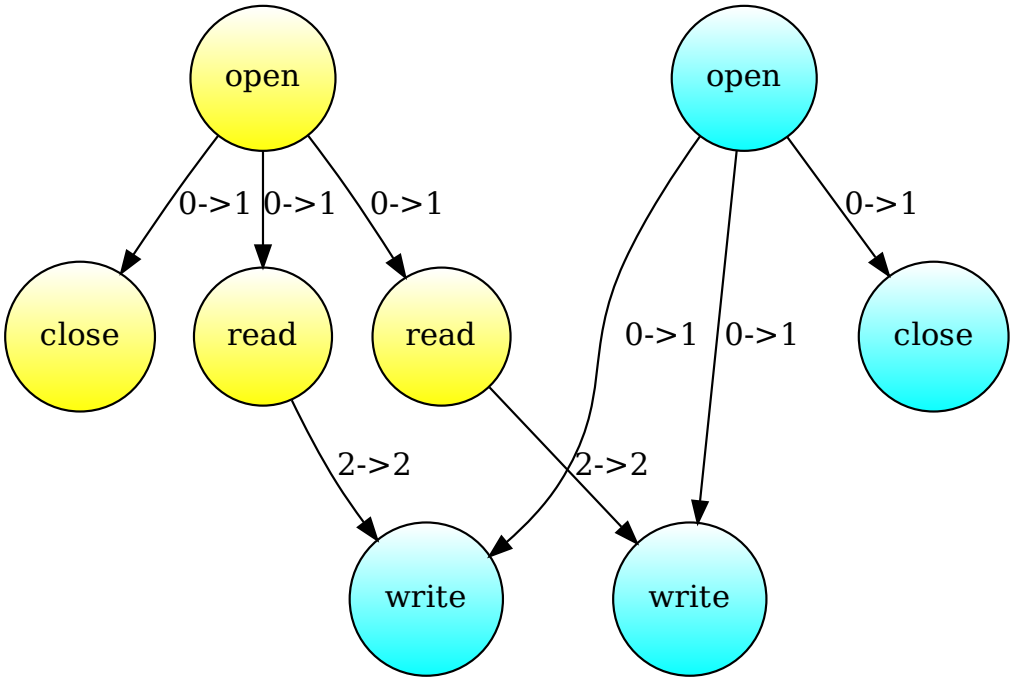

Figure 7: Ideal graph for the code in figure 5

Definition 6 (Argument, System Call)**.** *Let define an object* Argument *as a value* name*.*

Let define an object SystemCall *as a tuple* $(name, Args, rValue)$ *where:*

- name *is the name of the system call;*
- Args *is the set of its arguments, define as Argument objects;*
- rValue *is the return value of the System Call, define as an Argument object.*

We now formally define what is a direction of an argument. It is a function that associate a direction to each argument of a system call. The argument is referred to by its index.

Definition 7 (Direction of an Argument)**.** *A direction on S is a function*

$$dir : \mathbb{N} \times names(S) \to \{ Input, Output \}$$

with

- $dir(0, name(s)) = Output$*: the return value is always an Output;*
- $dir(n, name(s))$ *is well defined if* $n < card(Args(s))$*: the function is not defined if an index does not correspond to an argument.*

This definition does not specify how to construct the dir function. This will be done in a future work. Moreover, Microsoft provides such annotations for argument direction [9].

Example 2. *Let's consider the SystemCall* $(read, A_r, r_r)$*. We obtain:*

- $dir(0, \mathit{read}) =$ *Output*
- $dir(1, \mathit{open}) =$ *Input : it is* fd*;*
- $dir(2, \mathit{open}) =$ *Output : it is* buf*;*
- $dir(3, \mathit{open}) =$ *Input : it is* count*.*

Parameter buf *is considered as Output because the information flow*

Data cannot flow from an input or to an output argument. For example, a read does not modify the file described by the fd argument, so there is no information flow to the argument and no edge ending in this argument. In order to keep only consistent edges, the direction preprocessing keeps only the edges corresponding to flow from an Output to an Input argument.

Definition 8 (Direction preprocessing)**.** *Given a graph* $\mathcal{G}$*, and a direction function dir, we define the consistency preprocessing* PP_{dir} *as* (V', E', L', l') *such that*

$$
\begin{aligned}
&V' = V,\\
&L' = L,\\
&E' = \{\, (u,v) \in E \mid dir(src(l(u,v)), l(u)) = O \wedge dir(dest(l(u,v)), l(v)) = I \,\},\\
&\quad \textit{where } src(x{\to}y) = x, dest(x{\to}y) = y\\
&l' = l \textit{ restricted to } E' : l\big|_{E'}
\end{aligned}
$$

The graph presented in Figure 8 is obtained after having preprocessed the result obtained with the code in Figure 4. It just has one edge more than the ideal graph presented in Figure 7.

5 Methodology and validation

5.1 Methodology of validation

5.1.1 Mirai data-set

All servers offering Internet services must be able to manage a large number of requests. However, their capabilities are limited and surpassing them will result in a stoppage of service. Thus, it is possible to put out a system by overloading its bandwidth. This type of attack is called "denial of service" (abbreviated *DoS*) and undermines the availability of the target.[8]

Such an attack, if carried out by a single device, is very easy to prevent by limiting the rate of requests from a given host (ie. no more than 5 requests per seconds). Then, the idea came up to perform distributed denial of service attacks (abbreviated *dDoS*) in which requests are sent by different devices. Therefore, the target can hardly distinguish an attack from a buzz causing a large number of consultations.

Mirai is a malicious software that can find vulnerable IOT devices using Linux and infect them. For this, the malware tries to connect in a system by testing a large number of pairs

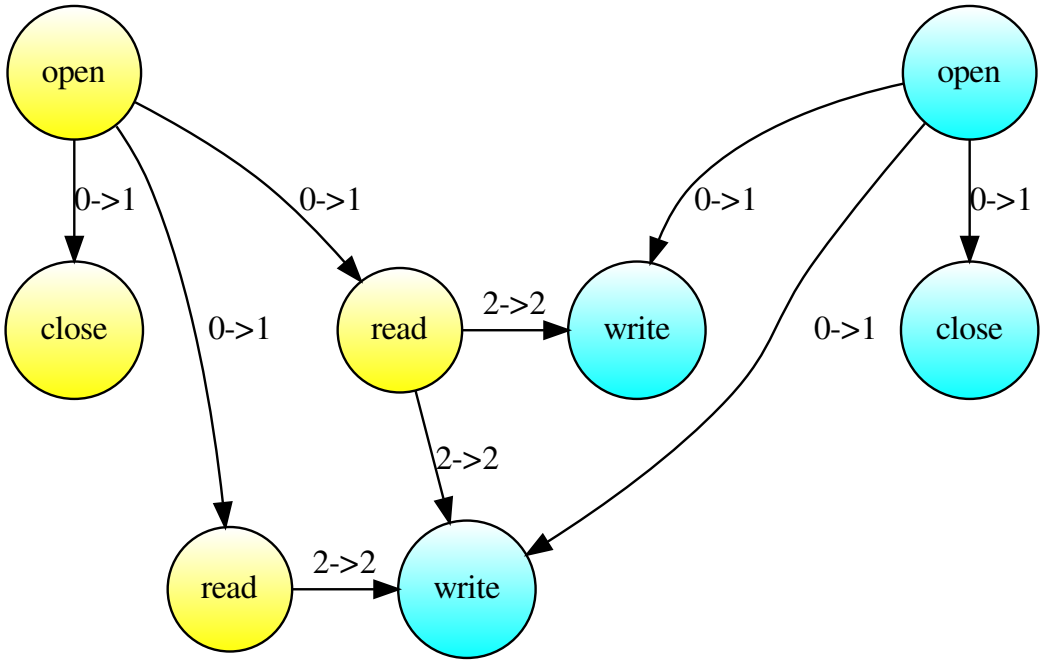

Figure 8: Graph obtained after preprocessing with code in Figure 5

username-password (see Appendix C). If it determines that it has successfully logged, it contacts its server to communicate the IP address, the TCP port and the pair username/password used [8]. *Mirai* created a botnet responsible for many attacks by *DoS* at the end of 2016 with exchanges up to 1 Tbps [6].

In this report, we use the graph data-set extracted in [8]. This data-set was obtained from Mirai samples captured using a honeypot and the static-get distribution.

Our database consists of 503 Mirais and 516 cleanwares. We chose that source of goodware because they are i386 static elf binaries, as our mirai samples are. The first part have been reduced by our method, but not the second. The five datasets consists on 403 Mirais and 403 cleanwares for the learning part, and 101 Mirais and 101 cleanwares for the testing one.

5.1.2 QuickSpan algorithm

QuickSpan is a very fast parallel implementation for gSpan algorithm in data mining [8]. It is used for the validation part of this report.

Quickspan is based on the gBolt [3] distributed implementation of gSpan. It extends gBold with the following features:

- taking edge direction into account (gSpan works on undirected graphs): there are two QuickSpan version, directed and undirected;

- stop mining after a given timeout (and return results found so far);
- specify minimum and maximum size of the graphs to output;
- stop mining when a graph of maximum size has reached the output.

Some of the features make this implementation more usable in practice (i.e. timeout). Some other allow to improve the performance. For instance, stopping as soon as we find a sub-graph large enough is very useful to speed up classification.

5.1.3 Experiments

To experiment our preprocessing, we validate it in two steps as in [8]. The first step aims to select the parameters. The second step is the actual evaluation with the parameters found at the previous step.

Parameter Exploration On a restricted number of datasets the parameters are varied: support (from 0.4 to 0.8 by increment of 0.1) and number of sub-graphs (1, 2, 5, 10, 25, 50) for gSpan, threshold (from 0.01 to 0.99 by increments of 0.01) for the classification, etc. We retain the parameters that give the best F0.5 score.

Validation With the best combination of parameters, we run the classification on 500 datasets formed by randomly selecting learning and testing sets among the available graphs. More precisely we used 80% of the available mirai samples for the learning set and the remaining 20% for the testing set. It was completed by randomly selecting goodware graphs to have the same number of mirai and goodware graphs.

Previous results Our results are compared with the results obtained by [8]. Two experiments were done.

The first one uses the undirected gSpan version, the second one the directed version and the parameters presented in Table 1. The learning data-sets contained 402 mirai graphs and the testing data-sets consist on 101 Mirais and 101 goodwares. The undirected version obtained the best $F_{0.5}$ score of 99.41%, with a learning time of 2,503s (approx. 12.4s per sample) and a classification time of 315.59s (approx. 1.56 seconds per sample).

Gspan Version	Support	# Subgraphs	Threshold	False Pos	False Neg	F-0.5 score
undirected	0.7	50	0.32	0.00%	2.88%	99.41%
directed	0.5	50	0.4	0.00%	7.08%	98.50%

Table 1: Parameters and results of the experiments in [8] over 500 experiments

5.2 Edges consistency method

5.2.1 Methodology

From the data-set of the *SCDG* extracted from *Mirai* samples (used in [8]) we extract the union of the *ET* of its graphs. Thereby, we obtain 190 edge types present in the *SCDGs* of *Mirai*. Then,

we create manually the *EC* function.

Example 3. *Let's consider the three edge types* $(fcntl, close, 1\text{->}1)$, $(fcntl, socketcall, 2\text{->}1)$ *and* $(fcntl, fcntl, 0\text{->}3)$.

In the first one, the two arguments concerned (first of fcntl *and* close*) are file descriptors. So these type of edges shows that the two calls involve the same file. This edge type is classified as* consistent*:*

$$EC((fcntl, close, 1\text{->}1)) = consistent$$

In the second one, the argument of fcntl *is* cmd *and the one of* scoketcall *is* call*. Such arguments are typically "small integers" defining what is the action made by the function. However, we don't have have a transmission of information from* fnctl *to* socketcall *since they represent different sub-commands of different calls. The edge type is classified as* meaningless*:*

$$EC((fcntl, socketcall, 2\text{->}1)) = meaningless$$

In the third one, the concerned arguments of fcntl *are its return value, which is expected to be a file descriptor, and* arg*, depending on the value of* cmd*. By studying the possible return value of this function, we see that it can be a file description (for example if* cmd=dupfd*) but not necessary. This edge type is classified as* indeterminate*:*

$$EC((fcntl, fcntl, 0\text{->}3)) = indeterminate$$

The result of this study is present in table 2 and by the appendix D.

Consistent	Meaningless	Indeterminate	Total
59	76	55	190

Table 2: Result of the classification of edge types

From this result, the graphs of Mirais are filtered: the consistent and indeterminate edges are kept, the meaningless one are deleted. This operation permits a reduction of the size of the graphs of 39%.

5.2.2 Results and validation

We validate this method by a 5-fold cross validation.

Parameter exploration The previous experiments [8] obtained with the directed gSpan version an average learning time of 2,503s (equal of the timeout) and an average classifying time of 315.59 seconds (for 202 graphs, so about 1.56s per graph). The table 3 presents our time results.

gSpan version	Learning	Classification	Classification per graph
undirected	8.36	454.04	2.27
directed	4.95	764.75	3.82

Table 3: Time results of the experiments (in s.)

This method permits to decrease the learning time, previously the timeout of 2,503 s., to 4.95 s. for the directed gSpan version. However, the classification time par graph increases with the preprocessing.

Table 4 reports the parameter combinations (support used by gSpan, threshold used for the classification and number of sub-graphs) with the best obtained $F_{0.5}$ score [1].

Gspan Version	Support	# Subgraphs	Threshold	False Pos	False Neg	F-0.5 score
directed	0.5	25	0.45 - 0.55	1.73%	2.49%	98.13%
directed	0.5	30	0.45 - 0.55	1.73%	2.49%	98.13%
directed	0.5	40	0.51 - 0.55	1.73%	2.49%	98.13%

Table 4: Parameter results of the experiments

This method, although faster, reduces the $F_{0.5}$ score of 1.28%. We consider that this reduction is too important regarding to the results obtained previously [8] and that it does not improve the classification time. We choose to not proceed with the second part of the validation.

Recall that goodware are not use in the learning phase (see Section 2.3). Recall that goodware graphs have not been reduced by our method (see Section 3.2). Building the *EC* function to annotate their edges might improve the average classification time.

5.3 Considering arguments direction

5.3.1 Methodology

For each of the system calls used by the programs in our data-set, we manually annotated the direction of the arguments. We obtained a list of 108 annotated system calls. Then, we apply direction preprocessing.

By noting $\mathcal{N}(G) = \frac{\text{number of edges of filtered}(G)}{\text{number of edges of } G}$ and $\overline{\mathcal{N}(G)}_{G\in A}$ its mean on the set A, we obtain:

$$\overline{\mathcal{N}(G)}_{G\in \text{ Mirais}} = 24.87\%$$
$$\overline{\mathcal{N}(G)}_{G\in \text{ Goodwares}} = 6.69\%$$
$$\overline{\mathcal{N}(G)}_{G\in \text{ Mirais} \cup \text{ Goodwares}} = 13.02\%$$

Moreover, some goodwares of our database become empty after this filtering.

5.3.2 Results and validation

Parameter exploration Our database is reduced on the nonempty graphs: 500 Mirais and 204 goodwares. To explore the parameters, 20 test run sets were randomly created, consisting of 400 Mirais samples for the learing phase and 100 Mirais and 100 goodware for the testing part (so a testing set of 200 samples).

Table 5 reports the parameter combinations (support, threshold and number of sub-graphs) with the best obtained $F_{0.5}$ score.

[1] $F_{0.5}$ score $= \left(1 + \left(\frac{1}{2}\right)^2\right) \frac{\text{precision.recall}}{\left(\frac{1}{2}\right)^2.\text{precision}+\text{recall}} = 5\frac{\text{precision.recall}}{\text{precision}+4\text{recall}}$

Gspan Version	Support	# Subgraphs	Threshold	False Pos	False Neg	F-0.5 score
directed	0.8	1	0.29 - 0.42	0.40%	2.55%	99.16%
undirected	0.8	1	0.29 - 0.42	0.40%	2.55%	99.16%
directed	0.8	2	0.34 - 0.42	0.40%	2.55%	99.16%
undirected	0.8	2	0.34 - 0.42	0.40%	2.55%	99.16%
directed	0.8	5	0.34 - 0.50	0.40%	2.55%	99.16%
undirected	0.8	5	0.34 - 0.50	0.40%	2.55%	99.16%
directed	0.8	10	0.41 - 0.50	0.40%	2.55%	99.16%
undirected	0.8	10	0.41 - 0.50	0.40%	2.55%	99.16%

Table 5: Parameter results of the experiments

We use 10 common sub-graphs because we want more stability that with a single sub-graph. To choose between directed and undirected, we compare the average execution time for both in Table 6. So we choose to use the undirected version.

Version	Learning time	Classification Time
directed	1.89s	30.38s
undirected	1.84s	24.10s

Table 6: Time results for the parameter exploration

Validation Moreover, we run the experiments on 500 data-sets with the undirected version of QuickSpan. The results are presented in table 7.

Metrics:

False positive rate	False negative rate
0.49%	2.41%
Recall	Precision
97.59%	99.50%
Specificity	F-0.5 score
99.51%	99.11%

Timings:

Learning	Classification
1.77 s	2.56 s

Table 7: Parameter results of the experiments with 500 data-sets

Here, we obtain a $F_{0.5}$ score of 99.11%, to compare with the 99.41% obtained previously [8]. Despite the reduction of the score, the timings are been improved: a speedup of 1,414x for learning and 123x for classification.

Note that empty SCDGs for goodware were not considered. We still need to understand why no edges were kept in these graphs. However since no edges were kept, they cannot introduce false positives, because none of their edges would match the ones kept in the mirai graphs.

6 Conclusion

This report presents two graph preprocessing methods to improve the analysis of *System Call Dependency Graphs*, created for malware detection. The two methods consists of the analysis

of the SCDG edge consistency and of the definition of a direction for the arguments of system calls. These methods have been experimentally validated. The first one does not give very satisfying results (i.e. lower the $F_{0.5}$ score by 1% compared to experiment in [8]) but is still faster. The second method give better results than the first one (i.e. loses only 0.3 points of $F_{0.5}$ score in [8]) and achieves a considerable speedup compared to [8]. This advocates for more research on preprocessing of the graphs when considering SCDGs-based malware classification.

Future works include comparing the results of preprocessing with taint analysis. Taint analysis is currently not implemented in the malware detection from [8], based on angr. Using taint analysis, we would be able to track the flow of information between calls, and directly output dependency information. An interesting study would be to compare graphs obtained from taint analysis with graphs obtained from the method presented in this report. We can also consider graphs obtained by mixing the two methods.

Another future work direction is to apply this work for windows malware, notably the second method. Since Microsoft provides such annotation for argument direction, the second method can be directly applied [9].

References

[1] Fabrizio Biondi, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. "Tutorial: an Overview of Malware detection and Evasion Techniques".

[2] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. "Mining specifications of malicious behavior". In: *ISEC*. ACM, 2008, pp. 5–14.

[3] *gBolt*. Accessed: 2018-08-23. URL: `https://github.com/Jokeren/DataMining-gSpan`.

[4] Fatemeh Karbalaie, Ashkan Sami, and Mansour Ahmadi. "Semantic malware detection by deploying graph mining". In: *International Journal of Computer Science Issues* 9.1 (2012), pp. 373–379.

[5] *Malware statistics*. Accessed: 2018-08-23. URL: `https://www.av-test.org/en/statistics/malware/`.

[6] *Record-breaking DDoS reportedly delivered by >145k hacked cameras*. Accessed: 2018-08-23. URL: `https://arstechnica.com/information-technology/2016/09/botnet-of-145k-cameras-reportedly-deliver-internets-biggest-ddos-ever/`.

[7] Kevin A. Roundy and Barton P. Miller. "Binary-code Obfuscations in Prevalent Packer Tools". In: *ACM Comput. Surv.* 46.1 (July 2013), 4:1–4:32. ISSN: 0360-0300. DOI: `10.1145/2522968.2522972`. URL: `http://doi.acm.org/10.1145/2522968.2522972`.

[8] Najah Ben Said, Fabrizio Biondi, Vesselin Bontchev, Olivier Decourbe, Thomas Given-Wilson, Axel Legay, and Jean Quilbeuf. "Detection of Mirai by Syntactic and Behavioral Analysis". In: *29th IEEE International Symposium on Software Reliability Engineering, ISSRE 2018, October 15-18, 2018*. Accepted, to appear. IEEE Computer Society, 2018.

[9] *Understanding SAL*. Accessed: 2018-08-23. URL: `https://msdn.microsoft.com/en-us/library/hh916383.aspx`.

[10] *x86 Linux Networking System Calls: Socketcall*. Accessed: 2018-08-23. URL: `http://jkukunas.blogspot.com/2010/05/x86-linux-networking-system-calls.html`.

[11] Xifeng Yan and Jiawei Han. “gSpan: Graph-Based Substructure Pattern Mining”. In: *Proceedings of the 2002 IEEE International Conference on Data Mining* (2002).

Appendix

A Some useful synopsis of system calls

Here is the synopsis of some system calls used in this report.

- **int** open(**const char** *pathname, **int** flags);
  int open(**const char** *pathname, **int** flags, mode_t mode);
- ssize_t read(**int** fd, **void** *buf, size_t count);
- ssize_t write(**int** fd, **const void** *buf, size_t count);
- **int** close(**int** fd);
- **int** ioctl(**int** fd, **unsigned long** request, ...);
- **int** socketcall(**int** call, **unsigned long** *args);

B Proof of the number of edges for n cycles read write

We consider a code with n cycles read write, like presented in Figure 2 (one cycle) and Figure 4 (two cycles). We want to know the number of edges $E(n)$ of its *System Call Dependency Graph*. We denote by open_A and close_A (respectively open_B and close_B) the open and close calls for the first file (respectively for the second file).

With our examples, we saw that:

$$\begin{aligned} E(0) &= 2 \\ E(1) &= 9 \\ E(2) &= 30 \end{aligned}$$

Let's proof that $E(n) = 7n^2 + 2$.

We have:

$$
\begin{aligned}
\forall n \in \mathbb{N}, E(n+1) = \quad & E(n) \\
& \left.\begin{aligned}
&+ \underbrace{1}_{open_A \to read_{n+1}} + \underbrace{1}_{open_B \to write_{n+1}} \\
&+ \underbrace{1}_{read_{n+1} \to close_A} + \underbrace{1}_{write_{n+1} \to close_B} \\
&+ \underbrace{n}_{read_k \to read_{n+1}} + \underbrace{n}_{write_k \to write_{n+1}}
\end{aligned}\right\} \text{exchange of file descriptior: f or f2} \\
& \left.\begin{aligned}
&+ \underbrace{n}_{read_k \to read_{n+1}} + \underbrace{n}_{write_k \to write_{n+1}} \\
&+ \underbrace{n}_{write_k \to read_{n+1}} + \underbrace{n}_{read_k \to write_{n+1}} \\
&+ \underbrace{1}_{read_{n+1} \to write_{n+1}}
\end{aligned}\right\} \text{exchange of buf with } k \in [\![1, n]\!] \\
& \left.\begin{aligned}
&+ \underbrace{n}_{read_k \to read_{n+1}} + \underbrace{n}_{write_k \to write_{n+1}} \\
&+ \underbrace{n}_{write_k \to read_{n+1}} + \underbrace{n}_{read_k \to write_{n+1}} \\
&+ \underbrace{1}_{read_{n+1} \to write_{n+1}}
\end{aligned}\right\} \text{exchange of count with } k \in [\![1, n]\!] \\
& \left.\begin{aligned}
&+ \underbrace{n}_{read_k \to read_{n+1}} + \underbrace{n}_{write_k \to write_{n+1}} \\
&+ \underbrace{n}_{write_k \to read_{n+1}} + \underbrace{n}_{read_k \to write_{n+1}} \\
&+ \underbrace{1}_{read_{n+1} \to write_{n+1}}
\end{aligned}\right\} \text{arrow from return value to count } k \in [\![1, n]\!]
\end{aligned}
$$

Then,

$$\forall n \in \mathbb{N}, E(n+1) = E(n) + 14n + 7$$

Therefore,

$$
\begin{aligned}
\forall n \in \mathbb{N}, E(n) - E(0) &= \sum_{k=0}^{n-1} (E(k+1) - E(k)) \\
&= \sum_{k=0}^{n-1} (14k + 7) \\
&= 14 \sum_{k=0}^{n-1} k + 7 \sum_{k=0}^{n-1} 1 \\
&= 14 \frac{n(n-1)}{2} + 7n \\
&= 7n^2
\end{aligned}
$$

So,

$$
\begin{aligned}
\forall n \in \mathbb{N}, E(n) &= 7n^2 + E(0) \\
&= 7n^2 + 2
\end{aligned}
$$

C Username/password pair list used by the Mirai Bot.

666666	666666
888888	888888
admin	
admin	1111
admin	1111111
admin	1234
admin	12345
admin	123456
admin	54321
admin	7ujMko0admin
admin	admin
admin	admin1234
admin	meinsm
admin	pass
admin	password
admin	smcadmin
admin1	password
administrator	1234
Administrator	admin
guest	12345
guest	guest
mother	fucker
root	
root	00000000
root	1111
root	1234
root	12345
root	123456
root	54321
root	666666
root	7ujMko0admin
root	7ujMko0vizxv
root	888888
root	admin
root	anko
root	default
root	dreambox
root	hi3518
root	ikwb
root	juantech
root	jvbzd
root	klv123
root	klv1234
root	pass
root	password
root	realtek
root	root
root	system
root	user
root	vizxv
root	xc3511
root	xmhdipc
root	zlxx.
root	Zte521
service	service
supervisor	supervisor
support	support
tech	tech
ubnt	ubnt
user	user

D Edge Consistency

D.1 Consistent edges

D.1.1 File descriptor

The following system calls dependencies exchange a file descriptor. There are the edges between:

- open, with 0 as argument;
- write, with 1 as argument;
- close, with 1 as argument;
- ioctl, with 1 as argument;
- newfstat, with 1 as argument;
- read, with 1 as argument;
- fcntl & fcntl64, with 1 as argument.

Table 8 presents the results.

D.1.2 Process ID

The following system calls dependencies exchange a file descriptor. There are the edges between:

- getpid, with 0 as argument;
- getppid, with 0 as argument;
- kill, with 1 as argument.

Table 9 presents the results.

D.1.3 Same SC with same argument

Two system calls which exchange the same argument have to be considered as linked. Table 10 presents the results.

D.1.4 Consistent in some cases (depending on the value of an argument)

Table 11 presents the results.

D.1.5 Other cases

Table 12 presents the results.

D.2 Meaningless edges

Tables 13 and 14 present the results.

D.3 Indeterminate edges

These edges have to be considered as indeterminate, because the first study did not treat them. Sometimes, this is caused by the different possible interpretation, depending on argument values.

The system call prctl was not studied. Tables 15 and 16 present the results.

Node 1	Argument	Node 2	Argument
sys_close	1	sys_close	1
sys_close	1	sys_fcntl	1
sys_close	1	sys_newfstat	1
sys_fcntl	1	sys_close	1
sys_fcntl	1	sys_fcntl	1
sys_fcntl64	1	sys_close	1
sys_fcntl64	1	sys_fcntl	1
sys_fcntl64	1	sys_fcntl64	1
sys_ioctl	1	sys_close	1
sys_ioctl	1	sys_fcntl	1
sys_ioctl	1	sys_newfstat	1
sys_ioctl	1	sys_write	1
sys_newfstat	1	sys_close	1
sys_newfstat	1	sys_fcntl	1
sys_open	0	sys_close	1
sys_open	0	sys_fcntl	1
sys_open	0	sys_ioctl	1
sys_open	0	sys_newfstat	1
sys_open	0	sys_write	1
sys_read	1	sys_read	1
sys_write	1	sys_close	1
sys_write	1	sys_read	1
sys_write	1	sys_write	1

Table 8: Consistent edges with exchange of file descriptor

Node 1	Argument	Node 2	Argument
sys_getpid	0	sys_kill	1

Table 9: Consistent edges with exchange of process id

Node 1	Argument	Node 2	Argument	Note
sys_fcntl	2	sys_fcntl	2	These system calls exchange a parameter for the type of operation (cmd, call, request, etc.)
sys_fcntl64	2	sys_fcntl64	2	
sys_ioctl	2	sys_ioctl	2	
sys_ipc	1	sys_ipc	1	
sys_socketcall	1	sys_socketcall	1	
sys_ipc	2	sys_ipc	2	These system calls exchange the same argument value.
sys_read	3	sys_read	3	
sys_rt_sigaction	1	sys_rt_sigaction	1	
sys_write	3	sys_write	3	
sys_read	2	sys_read	2	These system calls exchange the same argument pointer.
sys_rt_sigaction	2	sys_rt_sigaction	2	
sys_rt_sigaction	2	sys_rt_sigaction	3	
sys_rt_sigaction	3	sys_rt_sigaction	3	
sys_rt_sigprocmask	2	sys_rt_sigprocmask	2	
sys_rt_sigprocmask	3	sys_rt_sigprocmask	3	
sys_select	2	sys_select	2	
sys_select	5	sys_select	5	
sys_socketcall	2	sys_socketcall	2	
sys_times	1	sys_times	1	
sys_write	2	sys_write	2	
sys_futex	1	sys_futex	1	
sys_futex	2	sys_futex	2	
sys_futex	3	sys_futex	3	
sys_futex	5	sys_futex	5	
sys_futex	6	sys_futex	6	

Table 10: Consistent edges with exchange the same argument for same system calls

Node 1	Argument	Node 2	Argument	Condition
sys_fcntl	0	sys_fcntl	3	
sys_fcntl	2	sys_select	1	
sys_fcntl64	0	sys_fcntl64	3	cmd = dupfd*
sys_fcntl64	2	sys_select	1	
sys_write	1	sys_fcntl	3	

Table 11: Consistent edges in some cases

Node 1	Argument	Node 2	Argument	Note
sys_open	1	sys_readlink	1	
sys_read	2	sys_write	2	
sys_read	3	sys_write	3	
sys_rt_sigaction	1	sys_kill	2	
sys_rt_sigprocmask	2	sys_rt_sigprocmask	3	
sys_rt_sigprocmask	3	sys_rt_sigprocmask	2	

Table 12: Consistent edges – other cases

Name	Type	Note
fd	int	file descriptor
cmd	int	describes what to do in a fcntl
count	size_t	number of bits to write
flags	int	
call	int	describes the socket action to do (e.g. bind, connect, recv, etc.)
nb	int	number of written bits
how	int	describes what to do

Node 1	Argument	Sense	Node 2	Argument	Sense
sys_close	0	success	sys_fcntl	3	?
sys_close	0	success	sys_fcntl64	3	?
sys_close	1	fd	sys_rt_sigaction	1	signum
sys_close	1	fd	sys_rt_sigprocmask	1	how
sys_close	1	fd	sys_socketcall	1	call
sys_close	1	fd	sys_fcntl	2	cmd
sys_close	1	fd	sys_fcntl64	2	cmd
sys_close	1	fd	sys_open	2	flags
sys_close	1	fd	sys_read	3	count
sys_fcntl	2	cmd	sys_close	1	fd
sys_fcntl	2	cmd	sys_socketcall	1	call
sys_fcntl64	1	fd	sys_socketcall	1	call
sys_fcntl64	2	cmd	sys_close	1	fd
sys_fcntl64	2	cmd	sys_socketcall	1	call
sys_getrlimit	1	ressource	sys_close	1	fd
sys_getrlimit	1	ressource	sys_ioctl	1	fd
sys_getrlimit	1	ressource	sys_socketcall	1	call
sys_getrlimit	1	ressource	sys_fcntl	2	cmd
sys_ioctl	1	fd	sys_rt_sigaction	1	signum
sys_ioctl	1	fd	sys_rt_sigprocmask	1	how
sys_ioctl	1	fd	sys_socketcall	1	call
sys_ioctl	1	fd	sys_fcntl	2	cmd
sys_ioctl	1	fd	sys_fcntl64	2	cmd
sys_ioctl	1	fd	sys_read	3	count
sys_ioctl	1	fd	sys_write	3	count
sys_newfstat	1	fd	sys_fcntl	2	cmd
sys_open	0	fd	sys_rt_sigaction	1	signum
sys_open	0	fd	sys_socketcall	1	call
sys_open	0	fd	sys_fcntl	2	cmd
sys_open	0	fd	sys_fcntl64	2	cmd
sys_open	2	flags	sys_close	1	fd
sys_open	2	flags	sys_rt_sigaction	1	signum
sys_open	2	flags	sys_rt_sigprocmask	1	how
sys_open	2	flags	sys_socketcall	1	call
sys_open	2	flags	sys_fcntl	2	cmd

Table 13: Meaningless edges – part 1

Node 1	Argument	Sense	Node 2	Argument	Sense
sys_prctl	1	option	sys_write	3	count
sys_read	0	nb	sys_read	3	count
sys_read	0	nb	sys_write	3	count
sys_rt_sigprocmask	1	how	sys_close	1	fd
sys_rt_sigprocmask	1	how	sys_socketcall	1	call
sys_rt_sigprocmask	1	how	sys_write	1	fd
sys_rt_sigprocmask	1	how	sys_fcntl	2	cmd
sys_socketcall	0	? (sockfd ou nb)	sys_close	1	fd
sys_socketcall	0	? (sockfd ou nb)	sys_fcntl	1	fd
sys_socketcall	0	? (sockfd ou nb)	sys_fcntl64	1	fd
sys_socketcall	0	? (sockfd ou nb)	sys_read	1	fd
sys_socketcall	0	? (sockfd ou nb)	sys_select	1	nfds
sys_socketcall	0	? (sockfd ou nb)	sys_write	1	fd
sys_socketcall	0	? (sockfd ou nb)	sys_fcntl	2	cmd
sys_socketcall	0	? (sockfd ou nb)	sys_fcntl64	2	cmd
sys_socketcall	1	call	sys_close	1	fd
sys_socketcall	1	call	sys_fcntl	1	fd
sys_socketcall	1	call	sys_fcntl64	1	fd
sys_socketcall	1	call	sys_newfstat	1	fd
sys_socketcall	1	call	sys_rt_sigaction	1	signum
sys_socketcall	1	call	sys_rt_sigprocmask	1	how
sys_socketcall	1	call	sys_write	1	fd
sys_socketcall	1	call	sys_fcntl	2	cmd
sys_socketcall	1	call	sys_fcntl64	2	cmd
sys_socketcall	1	call	sys_open	2	flags
sys_socketcall	1	call	sys_fcntl	3	?
sys_socketcall	1	call	sys_read	3	count
sys_socketcall	1	call	sys_write	3	count
sys_write	0	nb	sys_close	1	fd
sys_write	0	nb	sys_ioctl	1	fd
sys_write	0	nb	sys_prctl	1	option
sys_write	0	nb	sys_socketcall	1	call
sys_write	0	nb	sys_fcntl	2	cmd
sys_write	1	fd	sys_socketcall	1	call
sys_write	1	fd	sys_read	3	count
sys_write	1	fd	sys_write	3	count
sys_write	3	count	sys_close	1	fd
sys_write	3	count	sys_ioctl	1	fd
sys_write	3	count	sys_prctl	1	option
sys_write	3	count	sys_socketcall	1	call
sys_write	3	count	sys_fcntl	2	cmd

Table 14: Meaningless edges – part 2

Node 1	Argument	Sense	Node 2	Argument	Sense
sys_chdir	1	*path	sys_prctl	3	arg3
sys_chdir	1	*path	sys_prctl	4	arg4
sys_close	1	fd	sys_prctl	4	arg4
sys_fcntl	2	cmd	sys_rt_sigaction	1	signum
sys_fcntl	2	cmd	sys_prctl	4	arg4
sys_futex	3	val	sys_futex	5	*timeout
sys_futex	5	*timeout	sys_futex	3	val
sys_getpid	0	pid	sys_prctl	3	arg3
sys_getpid	0	pid	sys_prctl	4	arg4
sys_getppid	0	pid	sys_prctl	3	arg3
sys_getppid	0	pid	sys_prctl	4	arg4
sys_ioctl	1	fd	sys_prctl	4	arg4
sys_ioctl	3	?	sys_ioctl	3	?
sys_ioctl	3	?	sys_prctl	3	arg3
sys_ipc	0	?	sys_ipc	2	firstArg
sys_ipc	3	secondArg	sys_socketcall	1	call
sys_ipc	3	secondArg	sys_fcntl	2	cmd
sys_ipc	3	secondArg	sys_rt_sigaction	4	?
sys_ipc	3	secondArg	sys_rt_sigprocmask	4	sigsetsize
sys_kill	1	pid	sys_prctl	3	arg3
sys_kill	1	pid	sys_prctl	4	arg4
sys_open	0	fd	sys_prctl	4	arg4
sys_prctl	3	arg3	sys_close	1	fd
sys_prctl	3	arg3	sys_socketcall	1	call
sys_prctl	3	arg3	sys_write	1	fd
sys_prctl	3	arg3	sys_fcntl	3	?
sys_prctl	3	arg3	sys_write	3	count
sys_prctl	4	arg4	sys_socketcall	1	call
sys_rt_sigaction	1	signum	sys_prctl	3	arg3
sys_rt_sigaction	1	signum	sys_prctl	4	arg4
sys_rt_sigaction	3	*oldact	sys_read	2	*buf
sys_rt_sigaction	4	?	sys_ipc	3	secondArg
sys_rt_sigaction	4	?	sys_rt_sigaction	4	?
sys_rt_sigaction	4	?	sys_rt_sigprocmask	4	sigsetsize
sys_rt_sigprocmask	4	sigsetsize	sys_ipc	3	secondArg
sys_rt_sigprocmask	4	sigsetsize	sys_rt_sigaction	4	?
sys_rt_sigprocmask	4	sigsetsize	sys_rt_sigprocmask	4	sigsetsize
sys_select	1	nfds	sys_close	1	fd
sys_socketcall	1	call	sys_prctl	3	arg3
sys_socketcall	1	call	sys_prctl	4	arg4
sys_socketcall	2	*args	sys_nanosleep	1	*req
sys_socketcall	2	*args	sys_nanosleep	2	*rem
sys_socketcall	2	*args	sys_rt_sigprocmask	2	*set
sys_socketcall	2	*args	sys_rt_sigprocmask	3	*oldset
sys_time	0	time	sys_prctl	3	arg3
sys_time	0	time	sys_prctl	4	arg4

Table 15: Indeterminate edges – part 1

Node 1	Argument	Sense	Node 2	Argument	Sense
sys_write	0	nb	sys_fcntl	3	?
sys_write	0	nb	sys_read	3	count
sys_write	0	nb	sys_write	3	count
sys_write	0	nb	sys_prctl	4	arg4
sys_write	2	*buf	sys_read	2	*buf
sys_write	3	count	sys_fcntl	3	?
sys_write	3	count	sys_read	3	count
sys_write	3	count	sys_prctl	4	arg4

Table 16: Indeterminate edges – part 2